MIT/LCS/TR-430

GRAPH-THEORETIC TECHNIQUES FOR PARALLEL, DISTRIBUTED, AND SEQUENTIAL COMPUTATION

Serge A. Plotkin

September 1988

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA200989

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-430

5. MONITORING ORGANIZATION REPORT NUMBER(S)
N00014-87-K-825 and N00014-86-K-0593

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

6c. ADDRESS *(City, State, and ZIP Code)*
545 Technology Square
Cambridge, MA 02139

7b. ADDRESS *(City, State, and ZIP Code)*
Information Systems Program
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Blvd.
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Graph-Theoretic Techniques for Parallel, Distributed, and Sequential Computation

12. PERSONAL AUTHOR(S)
Plotkin, S.A.

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
September 1988

15. PAGE COUNT
177

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Graph algorithms, parallel computation, distributed computation, parallel graph coloring, symmetry-breaking, combinatorial optimization, shared-memory muiti- (cont.)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Parallel computation presents problems which are either nonexistent or trivial in the context of sequential computation. Thus, design of efficient algorithms for parallel and distributed computation requires development of new tools and techniques.

This thesis considers a number of fundamental problems that arise in the context of parallel and distributed computation and describes several graph-theoretic techniques to address these problems. It also presents several new insights into the structure of various combinatorial optimization problems. In particular, the thesis presents the following results:

- A novel algorithm for symmetry breaking in distributed and parallel computing environments that runs in O(log*n) time

(cont.)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE *(Include Area Code)*
(617) 253-5894

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1985—507-047

Unclassified

18. processors, atomicity, I/O automata

19.

- A new primitive data object, called a Sticky Bit. A polynomial number of atomic Sticky Bits are sufficient to convert a safe implementation of an arbitrary sequential object into an atomic one in a shared-memory multiprocessing environment.

- A technique for managing a global resource in a distributed network. In particular, the technique allows us to convert an arbitrary distributed algorithm into an algorithm that terminates if the resource consumption exceeds some given value C, but behaves exactly like the original one as long as the resource consumption stays below C/2. Moreover, the conversion increases the complexity of the converted algorithm only by $o(\log^2 n)$ amortized per participating node, where n is the number of such nodes.

- A parallel algorithm for solving the minimum spanning tree problem on a n-by-n mesh-connected computer that runs in $O(n)$ time. The algorithm is novel because it is based on reducing the minimum spanning tree problem to the problem of finding shortest paths.

- The first sublinear-time parallel algorithm for bipartite matching. The algorithm runs in $O(n^{2/3}\log^3 n)$ time on a graph of n vertices, and can and can be generalized to solve maximum flow and minimum-cost flow problems in unit capacity networks.

- The sequential algorithms for the generalized circulation problem (network flow with losses and gains) which are the first polynomial-time combinatorial algorithms for this problem. One algorithm runs in $O(n^2m^2\log^2 n \log B)$ time and the other runs in $O(n^2m^2\log n \log^2 B)$ time, where n is the number of nodes, m is the number of edges, and B is the largest integer used to represent capacities and gains, where gains are represented as ratios of integers.

Graph-Theoretic Techniques for Parallel, Distributed, and Sequential Computation

by

Serge A. Plotkin

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 1988
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

Parallel computation presents problems which are either nonexistent or trivial in the context of sequential computation. Thus, design of efficient algorithms for parallel and distributed computation requires development of new tools and techniques.

This thesis considers a number of fundamental problems that arise in the context of parallel and distributed computation and describes several graph-theoretic techniques to address these problems. It also presents several new insights into the structure of various combinatorial optimization problems. In particular, the thesis presents the following results.

- A novel algorithm for symmetry breaking in distributed and parallel computing environments that runs in $O(\log^* n)$ time.
- A new primitive data object, called a *Sticky Bit.* A polynomial number of atomic Sticky Bits are sufficient to convert a safe implementation of an arbitrary sequential object into an atomic one in a shared-memory multiprocessing environment.
- A technique for managing a global resource in a distributed network. In particular, the technique allows us to convert an arbitrary distributed algorithm into an algorithm that terminates if the resource consumption exceeds some given value C, but behaves exactly like the original one as long as the resource consumption stays below $C/2$. Moreover, the conversion increases the complexity of the converted algorithm only by $O(\log^2 n)$ amortized per participating node, where n is the number of such nodes.
- A parallel algorithm for solving the minimum spanning tree problem on a n-by-n mesh-connected computer that runs in $O(n)$ time. The algorithm is novel because it is based on reducing the minimum spanning tree problem to the problem of finding shortest paths.
- The first sublinear-time parallel algorithm for bipartite matching. The algorithm runs in $O(n^{2/3} \log^3 n)$ time on a graph of n vertices, and can be generalized to solve maximum flow and minimum-cost flow problems in unit capacity networks.

- Two sequential algorithms for the generalized circulation problem (network flow with losses and gains) which are the first polynomial-time combinatorial algorithms for this problem. One algorithm runs in $O(n^2m^2\log^2 n\log B)$ time and the other runs in $O(n^2m^2\log n\log^2 B)$ time, where n is the number of nodes, m is the number of edges, and B is the largest integer used to represent capacities and gains, where gains are represented as ratios of integers.

Keywords: Graph algorithms, parallel computation, distributed computation, parallel graph coloring, symmetry-breaking, combinatorial optimization, shared-memory multiprocessors, atomicity, I/O automata.

Thesis Supervisor: Charles E. Leiserson

Title: Assoriate Professor of Computer Science and Engineering

Acknowledgments

I would like to thank everyone who have contributed to the contents of this thesis. I am very grateful to my advisor, Charles Leiserson, who has introduced me to the area of parallel computation and provided the encouragement, guidance, and support during the course of this research. Each one of the long discussions in his office provided me with a new perspective on my research, making me realize what I was doing in a much broader context. He has greatly helped me both in technical and nontechnical matters and promptly dealt with all those mundane problems like financial support, equipment, etc., creating a comfortable, stimulating, and creative research atmosphere.

I would like to express my thanks to Andrew Goldberg, who has convinced me to start doing research in parallel algorithms and to join the TOC group of the Laboratory for Computer Science. His contagious excitement was the source of constant inspiration, and I am indebted by his numerous contributions to my research progress.

I am grateful to David Shmoys both for his constant encouragement and advice, and for his invaluable comments on various drafts of the papers which make up this thesis. I would also like to thank him for many stimulating discussions which greatly contributed to my professional development and helped me achieve high standards of academic scholarship.

Éva Tardos has introduced me to the area of combinatorial optimization and showed me that some sequential algorithms are as interesting as parallel ones, and even more so. Her patience in explaining to me the basics of combinatorial optimization deserves my deepest appreciation. From her I have learned how to transform a rough idea into a mathematically correct proof, without loosing the initial intuition. Collaborating with Éva was a challenging, stimulating, and, most of all, an enjoyable experience. I would also like to thank her for numerous suggestions which improved the presentation of the thesis.

Baruch Awerbuch has introduced me to the problems that arise in distributed systems. He has showed me how to define new concepts and how to look for new research directions. From him I learned that stating a problem is sometimes harder and more important than solving it. Baruch's humor kept me in good spirits and enhanced the unique atmosphere of the third floor.

I would like to thank Nancy Lynch who suggested I consider the problems associated with implementing shared objects in a shared-memory multiprocessor. Her numerous suggestions were instrumental in obtaining the results presented in Chapter 2. I would also like to thank Yehuda Afek, Alan Fekete, and Michael Merritt, whose comments on various drafts of Chapter 2 were very helpful.

I have spent the summer of 1987 in AT&T Bell Laboratories in Murray Hill, where I had the opportunity to benefit from discussions with Yehuda Afek, David Johnson, Michael Saks, Robert Tarjan, and Pravin Vaidya. I would especially like to express my appreciation to David Johnson who made great efforts to make that summer both productive and enjoyable.

This thesis is derived from several papers. Chapter 1 presents joint work with Andrew Goldberg and Greg Shannon, Chapter 3 presents joint work with Yehuda Afek, Baruch Awerbuch, and Michael Saks. Chapter 4 presents joint work with Bruce Maggs. Chapter 5 is derived from a joint paper with Andrew Goldberg and Pravin Vaidya. Chapter 6 presents joint work with Andrew Goldberg and Éva Tardos.

I was privileged to be part of the exciting and stimulating environment of the Theory of Computation (TOC) group. I would like to thank all the members of the group for making it a great place to work.

Last but not the least comes my family. I am sure that I would have never gotten thus far without the confidence and support of my parents, [redacted] and [redacted] Plotkin. Their love and encouragement kept me going during the most difficult times.

This research was supported in part by the Defense Advanced Research Projects Agency Contract N00014-87-K-825 and by the Office of Naval Research Contract N00014-86-K-0593.

Contents

Introduction 11

1 Parallel Symmetry-Breaking in Sparse Graphs 17

1.1 Introduction . . . 17
1.2 Preliminaries . . . 19
1.3 Coloring Rooted Trees . . . 20
1.4 Coloring Constant-Degree Graphs . . . 24
1.5 Coloring and Matching in Planar Graphs . . . 27
1.6 Lower Bounds . . . 33
1.7 Conclusions and Open Problems . . . 34

2 Sticky Bits and Universality of Consensus 37

2.1 Introduction . . . 37
2.2 Model . . . 39
2.3 Wait-Free Atomicity . . . 42
2.4 Sticky Bit . . . 47
2.5 Atomic Implementation of an Arbitrary Object. . . . 49
2.6 Universality of Sticky Bit . . . 51
2.7 Conclusions and Open Problems . . . 60

3 Local Management of a Global Resource in a Communication Network 63

3.1 Introduction . . . 63
3.2 Model . . . 65
3.3 Resource Controller – Definition . . . 66
3.4 The Basic Controller. . . . 67
3.5 Main Controller . . . 74

3.6 Applications 76
3.6.1 Dynamic Name Assignment 76
3.6.2 Distributed Bank 77
3.7 Conclusions 78

4 Minimum-Cost Spanning Tree as a Path-Finding Problem 79
4.1 Introduction 79
4.2 Minimum-cost spanning tree 80
4.3 Implementation on a mesh-connected computer 81

5 Sublinear-Time Parallel Algorithms for Matching and Related Problems 85
5.1 Introduction 85
5.2 Definitions and Notation 88
5.3 Maximal Node-Disjoint Paths 89
5.4 Bipartite Matching and Zero-One Flows 94
5.4.1 Goldberg-Tarjan Maximum Flow Algorithm 94
5.4.2 Bipartite Matching Algorithm 95
5.4.3 Zero-One Flow Algorithms 102
5.5 The Assignment Problem and Minimum-Cost Flows 107
5.5.1 Goldberg-Tarjan Minimum-Cost Flow Algorithm 108
5.5.2 The Assignment Problem 110
5.5.3 Zero-One Minimum-Cost Flows 117
5.6 Conclusions 118

6 Combinatorial Algorithms for the Generalized Circulation Problem 121
6.1 Introduction 121
6.2 Definitions and Background 124
6.2.1 Minimum-Cost Circulation Problem 124
6.2.2 The Generalized Circulation Problem 127
6.2.3 Decomposition of Generalized Pseudoflows 129
6.2.4 Alternative Formulations 131
6.3 The Restricted Problem 132
6.4 Vertex Labels and Equivalent Problems 135
6.4.1 Canonical Relabeling 136
6.5 Simple Algorithms 140

6.6 Algorithm MCF 143

6.6.1 Analysis of the Inner Loop of the Algorithm 144

6.6.2 Bounding the Number of Iterations 146

6.7 The Fat-Paths Algorithm 151

6.7.1 Fat-Path Algorithm - Overview 152

6.7.2 Fat-Augmentation 155

6.7.3 Canceling Flow-Generating Cycles 158

6.7.4 Implementing the Generalized Dynamic Tree Data Structure 162

6.8 Conclusions 165

List of Figures

1.1 The Coloring Algorithm for Rooted Trees 21

1.2 The Coloring Algorithm for Constant Degree Graphs 25

1.3 The 7-Coloring Algorithm For Planar Graphs 28

2.1 Implementation of an object in terms of objects O and O' 42

2.2 Implementing Sticky Byte in terms of Sticky Bits 48

2.3 Atomic simulation of an object. 52

2.4 Information stored in a single cell 53

2.5 The *Grab* and *Release* procedures 54

2.6 The *Init* procedure. 55

2.7 The *GFC* procedure 56

2.8 The *Find-Head* procedure. 57

2.9 The *Append* procedure. 58

3.1 An example of a hierarchy on a chain 68

3.2 Description of the communication primitives. 70

3.3 The *Resource-Request* procedure 70

3.4 The *Delivery-Process* procedure 71

3.5 The *Root-Process* procedure 72

4.1 An example of a retimed mesh-connected computer 83

5.1 The *Maximal-Paths* procedure 90

5.2 Push and relabel operations. 95

5.3 The generic Goldberg-Tarjan maximum flow algorithm 96

5.4 High-level description of the bipartite matching algorithm 97

5.5 The *Match-and-Push* procedure 99

5.6 High-level description of the zero-one flow algorithm. 103

5.7 The *Push-and-Relabel* procedure. 104
5.8 The outer loop of the generic Goldberg-Tarjan minimum-cost flow algorithm 108
5.9 The inner loop of the generic Goldberg-Tarjan minimum-cost flow algorithm 109
5.10 Push and relabel operations for minimum-cost flow computation. 110
5.11 High-level description of the outer (scaling) loop of the assignment algorithm. . . . 111
5.12 High-level description of the inner loop of the assignment algorithm. 112

6.1 A single iteration of the maximum-flow based algorithm. 142
6.2 Inner loop of Algorithm MCF. 144
6.3 A single phase of the Fat-Path algorithm. 153
6.4 The Fat-Augmentation algorithm. 155
6.5 A single phase of the *Cancel-Cycles* algorithm. 160
6.6 Dynamic tree operations. 161
6.7 Generalized Dynamic Tree operations. 162
6.8 Implementation of Step 2 of the *Cancel-Cycles* algorithm. 163

Introduction

Parallel computers present the algorithm designer with many new challenges. Numerous problems which are encountered when designing parallel and distributed algorithms are either trivial or just nonexistent in the context of sequential computation. Thus, it is important to identify and isolate such problems, defining new paradigms, and devise general techniques to solve these abstracted problems.

The thesis identifies several core problems which one encounters when designing parallel and distributed algorithms and describes a variety of general techniques to address these problems. The thesis also presents numerous examples that show how to use these techniques to construct new algorithms or to improve the complexity of existing ones.

Efficient parallelization of a sequential algorithm usually requires new insight into the combinatorial structure of the problem. This, in turn, might lead to better sequential algorithms. Combinatorial optimization is an example of an area where an improvement in algorithm efficiency (both parallel and sequential) directly translates into our ability to solve larger problems. While the initial chapters of the thesis present the more general techniques which are applicable when designing a wide variety of parallel and distributed algorithms, the last two chapters concentrate on describing some new insights into the structure of several network-flow type problems, and illustrate the importance of these insights by describing several new parallel and sequential algorithms for combinatorial optimization problems.

The thesis is organized into 6 self-contained chapters. Following is a short overview of the ideas presented in each one of the chapters.

Symmetry-Breaking Some trivial sequential algorithms seem inherently unsuitable for parallel implementation. The problem often arises from the fact that these algorithms process information in small pieces at a time, where the way the next piece is processed depends on some of the previously processed ones. Allocating a processor per each unit of information leads to a situation in which we have many symmetric processors, where interdependencies prevent us from scheduling all of them at the same time. Symmetry-breaking techniques enable us to select a large number of processors that can be scheduled simultaneously.

When designing distributed algorithms, efficient symmetry-breaking techniques help decompose the network into small-radius regions. This decomposition is the heart of implementing divide-and-conquer algorithms in the distributed environment.

Chapter 1 describes a parallel technique to break symmetry in sparse graphs. This technique allows us to 3-color a rooted tree in $O(\log^* n)$ time on an EREW PRAM using a linear number of processors. The same techniques lead to an $O(\log^* n)$-time distributed algorithm for the same problem. This chapter shows how to use this technique and presents fast linear-processor algorithms for several problems, including the problem of $(\Delta + 1)$-coloring constant-degree graphs and 5-coloring planar graphs. Both the complexity and the number of processors used by these algorithms are significantly better than those of the previously known algorithms.

Shared Objects The usual way to implement shared data objects in the context of a shared-memory multiprocessor is to use synchronization primitives to "lock" the object in order to ensure that only a single processor accesses it at any given moment. The disadvantage of this approach is that failure of a processor that "has the lock" may prevent others from accessing the object, leading to a situation in which no processor can make any progress. Moreover, even if the processor that has the lock does not fail but is just slow, the faster processors may have to wait until the slower one finishes its access, which, in turn, reduces the overall performance of the system.

Chapter 2 introduces a general technique that allows wait-free implementation of any sequential object. The heart of this technique is a new primitive object, the "Sticky-Bit", which can be easily implemented in hardware or simulated by a randomized algorithm directly from safe bits. Using this object it is easy to make asynchronous processors "help" each other, so that even if a processor fail-stops in the middle of an operation which has to be completed before any other operation can be started, other processors will "help" him to complete this operation.

The Sticky Bit may be viewed as a memory-oriented generalization of consensus. In particular, the results of this chapter imply "universality of consensus" in the sense that given an algorithm to achieve n-processor consensus, we can transform any safe implementation of a sequential object into a wait-free atomic one.

Distributed resource management Often, when designing a distributed algorithm, we are faced with the situation where we are given a protocol that has bounded message complexity only under certain assumptions. Although the assumptions are usually reasonable, they might still be incorrect, and in this case it is desirable to transform the protocol, adding a mechanism that terminates it when the message complexity reaches dangerous levels. In a sense, we would like to "buy insurance", where the "premium" is the increased complexity of the protocol due to the additional mechanism, and the "deductible" is the maximum message complexity of the transformed protocol under any circumstances.

Chapter 3 considers a natural generalization of the problem of limiting the maximum message complexity, *i.e.* construction of a general *Resource Controller* that allows us to terminate or redirect a distributed algorithm according to the level of consumption of a global resource. In addition to the number of messages sent, a resource may be the total number of participating nodes, total CPU time consumed, total number of disk blocks in use, etc.

This chapter formalizes the notion of "Global Resource Controller" and describes a controller that achieves $O(\log^2 n)$ amortized message complexity per each request to use a unit of resource, where n is the number of participating nodes. This is a significant improvement over the $O(n)$ amortized message complexity of a naive controller. The chapter also describes how to use this controller to design efficient algorithms for several important problems, including the problem of approximating the size of a dynamically growing network and assigning small identification numbers (ID's) to nodes in such network.

Minimum-weight spanning tree as a path-finding problem Linear and mesh-connected systolic arrays are an important class of parallel computers. The simplicity of their interconnection network presents the algorithm designer with the challenge to make communication as local as possible. Consequently, efficient PRAM algorithms often do not translate into efficient algorithms for mesh-connected computers. In particular, consider the problem of finding a minimum-weight spanning tree of a given graph. Though algorithms that solve this problem are very simple in the context of PRAMs, direct translation of these algorithms to run on a mesh-connected computer leads to complicated recursive algorithms.

Chapter 4 shows that the problem of finding a minimum-weight spanning tree can be viewed as an instance of the path-finding problem in a closed semiring. This immediately implies a very simple linear-time algorithm for finding a minimum-weight spanning tree on a mesh-connected computer.

Sublinear-time algorithms for matching Bipartite matching is a well-studied problem in the contexts of both sequential and parallel computation. Though efficient randomized parallel algorithms for bipartite matching are known, the best parallel deterministic algorithm runs in superlinear time.

Chapter 5 describes a combinatorial technique that leads to the first sublinear-time deterministic parallel algorithm for bipartite matching and several related problems, including the problem of finding maximum flow in zero-one capacity networks, depth-first search, and the problem of finding a maximal set of node-disjoint paths. The technique can be applied to design parallel algorithms for the weighted versions of these problems. In particular, the chapter describes sublinear-time deterministic parallel algorithms for finding a minimum-weight bipartite matching and for finding a minimum-cost flow in a network with zero-one capacities, if the weights are polynomially bounded integers.

Generalized circulation problem Consider a generalization of the maximum flow problem in which the amounts of flow entering and leaving an arc are linearly related. More precisely, if $x(e)$ units of flow enter an arc e, $x(e)\gamma(e)$ units arrive at the other end. For instance, nodes of the graph can correspond to different currencies, with the multipliers being the exchange rates. The goal is to maximize the amount of flow excess at the source, while maintaining flow-conservation constraints at every other node. The generalized circulation problem is important both from a practical point of view, because it can be used to model many problems in fields ranging from science and engineering to operations research, and from a theoretical point of view, because it seems to be the "simplest" linear program for which no strongly polynomial algorithm is known.

Chapter 6 presents the first polynomial-time combinatorial algorithms[1] for the generalized circulation problem. The importance of designing such algorithms for this problem is twofold. Combinatorial methods may lead to algorithms that are more efficient, both in theory and in practice, than algorithms based on general linear programming techniques. (For example, this is the case for the minimum-cost circulation problem.) Furthermore, it seems that that a combinatorial approach is more likely to yield the insight needed to design a strongly polynomial algorithm for the generalized circulation problem, and the presented results can be viewed as a significant first step in this direction.

[1] By combinatorial algorithms we mean algorithms that exploit the combinatorial structure of the underlying network (as opposed to being based on analytic ideas like the interior point methods for linear programming).

Chapter 1

Parallel Symmetry-Breaking in Sparse Graphs

1.1 Introduction

Some problems for which trivial sequential algorithms exist appear to be much harder to solve efficiently in a parallel or distributed framework. When converting a sequential algorithm to a parallel one, at each step of the parallel algorithm we have to choose a set of operations which may be executed in parallel. Often, we have to choose these operations from a large set of symmetrical operations, where interdependencies prevent simultaneous execution of all the operations in the set. Symmetry-breaking techniques enable the algorithm to select a large subset of independent operations.

Finding a maximal independent set (MIS) of a graph is a good example of the necessity of symmetry-breaking. At any step, a parallel MIS algorithm might have many candidate nodes to add to the independent set. Due to adjacency constraints, however, not all of these nodes can be added simultaneously. A symmetry-breaking technique is therefore needed to find a large set of nodes to add, as has been done in the first parallel MIS algorithm of Karp and Wigderson [78] as well as in the subsequent improvements by Luby [93] and Goldberg and Spencer [65].

Previous symmetry-breaking techniques have focused on randomization. It is often desirable, however, to have a deterministic algorithm. Karp and Wigderson [78], and Luby [93]

[0] This chapter describes joint research with A. Goldberg [50, 51]. This work was merged with results by G. Shannon and published in [53, 52].

proposed methods to convert certain randomized algorithms into deterministic ones. Their methods, however, significantly increase the number of processors used.

In many cases it is sufficient to break symmetry in sparse graphs. The only previously known efficient symmetry-breaking technique, due to Cole and Vishkin [31], works only for a directed chain. In this chapter, we generalize their approach to obtain a deterministic symmetry-breaking technique to 3-color a rooted tree in $O(\log^* n)$ time on a CREW PRAM. Using our techniques, we develop the linear-processor algorithms listed below. Our results improve the running rime and processor bounds for respective problems.

- For graphs whose maximum degree is Δ, we give an $O((\log\Delta)(\Delta^2+\log^* n))$-time EREW PRAM algorithm for $(\Delta+1)$-coloring and for finding a maximal independent set. The best previous deterministic linear-processor algorithm for finding MIS, due to Goldberg and Spencer [65], runs in $O(\log^4 n)$ time on constant-degree graphs.

- For planar graphs, we give 7-coloring, MIS, and maximal matching algorithms that run in $O(\log n)$ time on a CRCW PRAM. The $O(\log^3 n)$ running time of the maximal matching algorithm due to Israeli and Shiloach [72] can be reduced to $O(\log^2 n)$ in the restricted case of planar graphs, which is $\log n$ factor worse than our algorithm.

- We give an $O(\log n \log^* n)$-time CRCW PRAM algorithm for 5-coloring an embedded planar graph. In comparison, the 5-coloring algorithms for planar graphs of Boyar and Karloff [22] and of Naor [103] are based on $O(\log n)$ applications of an MIS subroutine. If Luby's MIS is used [93], these algorithms run in $O(\log^3 n)$ time using $O(n^3)$ processors; if Goldberg-Spencer MIS is used [65], the processor requirements can be reduced to $O(n)$ at the expense of increasing the running time to $O(\log^5 n)$.

Although in this chapter we have limited ourselves to the application of our techniques for the design of parallel algorithms for the PRAM model of computation, the same techniques can be applied in a distributed model of computation [11, 46]. In particular, our algorithms for 3-coloring rooted trees and for $\Delta+1$ coloring can be applied directly, achieving $O(\log^* n)$ time. Moreover, the $\Omega(\log^* n)$ lower bound due to Awerbuch [13] and Linial [91] for the MIS problem on a chain in the distributed model implies that our symmetry-breaking technique is optimal in this model.

Since we can 3-color a rooted tree in $O(\log^* n)$ time in a PRAM model of computation, it is natural to ask if a rooted tree can be 2-colored as quickly. We answer this question by giving an $\Omega(\log n/\log\log n)$ lower bound for 2-coloring of a rooted tree. We also present an $\Omega(\log n/\log\log n)$ lower bound for finding a maximal independent set in a general graph, thus answering the question posed by Luby [93].

This chapter is organized as follows. Section 2 presents definitions, notation, and computation model details. Section 3 presents the algorithm for 3-coloring rooted trees. Section 4 uses this algorithm to $(\Delta + 1)$-color constant-degree graphs. Section 5 uses results of Section 4 to develop algorithms for planar graphs. Section 6 proves the lower bounds mentioned earlier. Section 1.7 summarizes the results and discusses some open problems.

1.2 Preliminaries

This section describes the assumptions about the computational model and introduces the notation used throughout the chapter. We consider simple, undirected graphs with n vertices and m edges. The maximum degree of a graph is denoted by Δ. The graph induced by a set of nodes X is denoted by $G[X]$.

We discuss the following problems:

- The node-coloring problem: find an assignment $C : V \rightarrow I^+ \cup \{0\}$ of nonnegative (not necessarily consecutive) integers (colors) to nodes of the graph so that no two adjacent nodes have the same color and so that the total number of colors is at most $\Delta + 1$. We assume that the bits are numbered from 0, and the ith bit in the color of a node v is denoted by $C_v(i)$.

- The maximal independent set (MIS) problem: a subset of nodes $I \subseteq V$ is *independent* if no two nodes in I are adjacent.

- The maximal matching (MM) problem: A subset of edges $M \subseteq E$ is a matching if each pair of distinct edges in M have no nodes in common.

We make a distinction between *unrooted* and *rooted* trees. In a rooted tree, each nonroot node knows which of its neighbors is its parent.

The following notation is used:

$$\begin{aligned}
\log x &= \log_2 x \\
\log^{(1)} x &= \log x \\
\log^{(i)} x &= \log(\log^{(i-1)} x) \\
\log^* x &= \min\{i \mid \log^{(i)} x \le 2\}
\end{aligned}$$

We assume a PRAM model of computation [21, 42] where each processor is capable of executing simple word and bit operations. The word width is assumed to be $O(\log n)$. The word operations we use include bit-wise boolean operations, integer comparisons, and unary-to-binary conversion. Each processor P has a unique *identification number* $O(\log n)$ bits wide, which we denote by ID(P). We use adjacency lists to represent the graph, assigning a processor to each edge and each node of the graph. We use exclusive-read exclusive-write (EREW) PRAM, concurrent-read exclusive-write (CREW) PRAM, and concurrent-read concurrent-write (CRCW) PRAM, as appropriate. The write conflicts in CRCW PRAM are assumed to be resolved arbitrarily. All lower bounds are proven for a CRCW PRAM with a polynomial number of processors.

1.3 Coloring Rooted Trees

This section presents a technique to 3-color a rooted tree in $O(\log^* n)$ time. This technique can be viewed as a generalization of a technique due to Cole and Vishkin [31] that finds a maximal independent set on a directed chain in $O(\log^* n)$ time. We first describe an $O(\log^* n)$-time algorithm for 6-coloring rooted trees and then show how to transform a 6-coloring of a rooted tree into a 3-coloring in constant time.

The procedure 6-Color-Rooted-Tree is shown in Figure 1.1. This procedure accepts a rooted tree $T = (V, E)$ and 6-colors it in time $O(\log^* n)$. Starting from the valid coloring given by the processor ID's, the procedure iteratively reduces the number of bits in the color descriptions by recoloring each nonroot node v with the color obtained by concatenating the

```
Procedure 6-COLOR-ROOTED-TREE($T$)
    $N_c \leftarrow n$;
    for all $v \in V$ in parallel do $C_v \leftarrow \mathrm{ID}(v)$;
    while $N_c > 6$ do
        for all $v \in V$ in parallel do begin
            if $v$ is the root then begin
                $i_v \leftarrow 0$;
                $b_v \leftarrow C_v(0)$;
            end;
            else begin
                $i_v \leftarrow \min\{i \mid C_v(i) \neq C_{parent(v)}(i)\}$;
                $b_v \leftarrow C_v(i_v)$;
            end;
            $C_v \leftarrow i_v b_v$;                    ⟨⟨ Bit $i_v$ concatenated with $b_v$ ⟩⟩
        end;
        $N_c \leftarrow \max\{C_v \mid v \in V\} + 1$;
    end;
end.
```

Figure 1.1: The Coloring Algorithm for Rooted Trees

index of a bit in which C_v differs from $C_{parent(v)}$ and the value of this bit. The root r forms its new color by concatenating 0 and $C_r[0]$.

Theorem 1.3.1 The algorithm 6-COLOR-ROOTED-TREE produces a valid 6-coloring of a tree in $O(\log^* n)$ time on a CREW PRAM using a linear number of processors.

Proof: First we prove by induction that the coloring computed by the algorithm is valid, and then we prove the upper bound on the execution time.

Assuming that the coloring C is valid at the beginning of an iteration, we first show that the coloring at the end of the iteration is also valid. Let v and w be two adjacent nodes with v being the parent of w. In the algorithm, w chooses some index i such that $C_v(i) \neq C_w(i)$, and v chooses some index j such that $C_v(j) \neq C_{parent(v)}(j)$. The new color of w is $\langle i, C_w(i)\rangle$, and the new color of v is $\langle j, C_v(j)\rangle$. If $i \neq j$, the new colors are different and we are done. On the other hand, if $i = j$, then $C_v(i)$ can not be equal to $C_w(i)$ by the definition of i, and again the colors are different. Hence, the validity of the coloring is preserved.

Now we use induction to show that the algorithm terminates after $O(\log^* n)$ iterations. Let L_k denote the number of bits in the representation of colors after k iterations. For $k = 1$ we have

$$L_1 = \lceil \log L \rceil + 1.$$

Assume for some k we have $L_{k-1} \leq \lceil \log^{(k-1)} L \rceil + 2$ and $\lceil \log^{(k)} L \rceil \geq 2$. Then $\lceil \log^{(k-1)} L \rceil \geq 4$ and we have

$$\begin{aligned} L_k &= \lceil \log L_{k-1} \rceil + 1 \\ &\leq \lceil \log(\lceil \log^{(k-1)} L \rceil + 2) \rceil + 1 \\ &\leq \lceil \log(\log^{(k-1)} L + 3) \rceil + 1 \\ &\leq \lceil \log(2(\log^{(k-1)} L)) \rceil + 1 \\ &\leq \lceil \log^{(k)} L \rceil + 2. \end{aligned}$$

Therefore, as long as $\lceil \log^{(k)} L \rceil \geq 2$,

$$L_k \leq \lceil \log^{(k)} L \rceil + 2.$$

Hence, the number of bits in the representation of colors L_k decreases until, after $O(\log^* n)$ iterations, $\lceil \log^{(k)} L \rceil$ becomes 1 and L_k reaches the value of 3. Another iteration of the algorithm produces a 6-coloring: 3 possible values of the index i_v and 2 possible values of the bit b_v. The algorithm terminates at this point.

Using concurrent-read, each node determines its parent's color in constant time. Given two colors, C_v and C_w, we can compute the smallest index j such that the jth bit of C_v differs from the jth bit of C_w by computing $j =$ unary-to-binary($|C_v - C_w|$ XOR $(|C_v - C_w| - 1)$). Hence, each node can compute the new color independently in constant time. Therefore, each iteration takes constant time and the algorithm uses $O(\log^* n)$ time overall. Note that no concurrent-write capabilities are required; for constant-degree trees the concurrent-read capability is not needed either. ▮

We now describe the algorithm 3-Color-Rooted-Tree which 3-colors a rooted tree. The algorithm first applies 6-Color-Rooted-Tree to produce a valid 6-coloring of the tree. Then it executes three stages, each time reducing the number of colors by one.

Each stage works as follows. By *shifting down* the coloring we mean recoloring each nonroot node with the color of its parent and recoloring the root with a color different from its current color. To remove the color $c \in \{3,4,5\}$, first shift down the current coloring. Then, recolor each node of color c with the smallest color different from its parent's and children's colors.

Theorem 1.3.2 Given a rooted tree T, the algorithm 3-Color-Rooted-Tree constructs a valid 3-coloring of T using n processors and $O(\log^* n)$ time on a CREW PRAM.

Proof: After a shift of colors, the children of any node have the same color. Thus each node is adjacent to nodes of at most two different colors. Therefore, each stage of the algorithm reduces the number of colors by one, as long as the number of colors is greater than three. Each stage takes a constant time on a CREW PRAM. The theorem follows from Theorem 1.3.1. ▌

To describe the subsequent algorithms, we introduce the concept of a pseudoforest [111]. A *pseudoforest* of $G = (V, E)$ is a directed graph $G' = (V, E')$, such that $(u, v) \in E' \Rightarrow \{u, v\} \in E$ and outdegree of any node is at most one. A *maximal pseudoforest* of $G = (V, E)$ is a directed graph $G' = (V, E')$, such that $(u, v) \in E' \Rightarrow \{u, v\} \in E$ and outdegree of any node in G' is one, unless this node is zero-degree in G. Nodes with zero out-degree are *roots* of the pseudoforest. We assume that graphs are represented by adjacency lists, and therefore a maximal pseudoforest can be constructed in (parallel) constant time by choosing an arbitrary adjacent edge for every node and directing this edge outward.

The coloring algorithms presented in this section work for pseudoforests as well as for rooted trees. Therefore, a pseudoforest can be 3-colored in $O(\log^* n)$ time on an CRCW PRAM using a linear number of processors. We shall call the procedure for 3-coloring pseudoforests 3-Color-Pseudoforest. Note that an odd cycle, which is a pseudoforest, can not be colored in less than 3 colors, and therefore the number of colors used by the procedure 3-Color-Pseudoforest is optimal in this case.

Any tree can be 2-colored. In fact, it is easy to 2-color a tree in polylogarithmic time. For example, one can use treefix operations [87, 101] to compute the distance from each node to the root, and color even level nodes with one color and odd-level nodes with the other color. It is harder to find a 2-coloring of a rooted tree in parallel, however, than it is to find a 3-coloring of a rooted tree. In Section 1.6 we show a lower bound of $\Omega(\log n / \log\log n)$ on 2-coloring of a directed list on a CRCW PRAM with a polynomial number of processors, which implies the same lower bound for rooted trees.

1.4 Coloring Constant-Degree Graphs

The algorithm for coloring rooted trees, described in the previous section, can be generalized to color constant-degree graphs in a constant number of colors [51]. In the generalized algorithm, a current color of a node is replaced by a new color obtained by looking at each neighbor, appending the index of a bit in which the current color of the node is different from the neighbors' color to the value of the bit in the node color, and concatenating the resulting strings. This algorithm runs in $O(\log^* n)$ time, but the number of colors, although constant as a function of n, is exponential in the degree of the graph.

In this section we show how to use the procedure 3-Color-Pseudoforest, described in the previous section, to color a constant-degree graph with $(\Delta+1)$ colors.

The algorithm Color-Constant-Degree-Graph, which colors a constant-degree graph $G = (V, E)$ with $(\Delta+1)$ colors, is presented in Figure 1.2. The algorithm consists of two phases. The first phase repeatedly constructs a maximal pseudoforest and removes its edges from G. This phase terminates when no edges remain, at which point all nodes are colored with one color. Then we color all the pseudoforests with 3 colors in parallel.

The second phase iteratively returns the edges of the current pseudoforest, each time recoloring the nodes to maintain a consistent coloring. At the beginning of each iteration of this phase, the edges E' of the current pseudoforest are added, making the existing $(\Delta + 1)$-coloring inconsistent. The forest E' is already colored with 3 colors. Now, each node has two colors: one from the coloring at the previous iteration and one from the coloring of the forest. The pairs of colors form a valid $3(\Delta+1)$-coloring of the graph. The iteration finishes by enumerating the color classes, recoloring each node of the current color with a color from $\{0, \ldots, \Delta\}$ that is different from the colors of its neighbors. We can recolor all the nodes of the same color in parallel because they are independent.

Theorem 1.4.1 *The algorithm* Color-Constant-Degree-Graph *colors the graph with* $(\Delta+1)$ *colors and runs in* $O((\log\Delta)(\Delta^2 + \log^* n))$ *time on an EREW PRAM using a linear number of*

```
Procedure COLOR-CONSTANT-DEGREE-GRAPH.
    $E' \leftarrow \{(v,w) \mid \{v,w\} \in E\}$;
    for $i = 0$ to $\Delta$ do begin                                  ⟨⟨ the first phase⟩⟩
        for all $v \in V$ in parallel do
            if $\exists (v,u) \in E'$ then $E_i \leftarrow E_i + (v,u)$ ;
        $E' \leftarrow E' - E_i$;                       ⟨⟨ $E_i$ are edges of a maximal pseudoforest ⟩⟩
    end;
    for all $v \in V$ in parallel do                                 ⟨⟨ initial coloring⟩⟩
        $C(v) \leftarrow 0$;
    for all $0 \leq i \leq \Delta$ in parallel do                      ⟨⟨ color the pseudoforests⟩⟩
        $C_i \leftarrow$ 3-COLOR-PSEUDOFOREST$(V, E_i)$;
    for $i \leftarrow \Delta$ down to 0 do begin                       ⟨⟨ the second phase⟩⟩
        $E' \leftarrow E' + E_i$;
        for $k \leftarrow 1$ to 2, $j \leftarrow 0$ to $\Delta$ do
            $V' \leftarrow V$;
            for all $v \in V'$ in parallel do
                if $C(v) = j$ and $C_i(v) = k$
                  then begin
                  $C(v) \leftarrow \max\{\{0,1,\ldots,\Delta\} - \{C(w) \mid (v,w) \in E'\}\}$;
                  $V' \leftarrow V' - \{v\}$;
                end;
            end;
        end;
    end;
end.
```

Figure 1.2: The Coloring Algorithm for Constant Degree Graphs

processors.

Proof: At each iteration all edges of the maximal pseudoforest are removed. The definition of a maximal pseudoforest implies that each node that still has neighbors in the beginning of an iteration has at least one edge removed during that iteration and therefore its degree decreases. After at most Δ iterations, E' is empty. The running time of each iteration is determined by the time required to select an unused edge out of an edge list. On an EREW PRAM, an unused edge can be selected in $O(\log \Delta)$ time. The pseudoforests are edge-disjoint and therefore can be colored in parallel. By Theorem 1.3.2, this takes $O(\log \Delta \log^* n)$ time on an EREW PRAM. The $\log \Delta$ factor appears because we do not use the concurrent-read capability; a node must broadcast its color to its children using, for example, parallel prefix computation [83, 101]. The total time bound for the first stage is therefore $O((\log \Delta)(\Delta + \log^* n))$.

The second phase terminates in at most Δ iterations as well, one iteration per pseudoforest. At each iteration we execute $O(\Delta)$ steps, that is, 2 steps for each color. Using integer sorting algorithm of Reif [113], each such step can be implemented in a straightforward way to run in $O(\log \Delta)$ time. For example, in order to find a color which is different from the colors of the neighbors, we can sort the colors of the neighbors and then use parallel prefix computation to find the first "gap". Hence, one iteration of the second phase takes $O(\Delta \log \Delta)$ time, leading to an overall $O((\log \Delta)(\Delta^2 + \log^* n))$ running time for the second stage of the algorithm and for the algorithm itself. ∎

Given a $(\Delta + 1)$-coloring of a graph, we can find an MIS of the graph.

Theorem 1.4.2 An MIS in constant-degree Δ graphs can be found in $O((\log \Delta)(\Delta^2 + \log^* n))$ time on an EREW PRAM using a linear number of processors.

Proof: Iterate over the colors, taking all the remaining nodes of the current color, adding them to the independent set, and removing them and all their neighbors from the graph. (We refer to this procedure as Constant-Degree-MIS in the subsequent sections.) The running time of this algorithm is dominated by the running time of the Color-Constant-Degree-Graph procedure. ∎

Remark: The proofs of Theorems 1.4.1 and 1.4.2 imply that the algorithms Color-Constant-

DEGREE-GRAPH and CONSTANT-DEGREE-MIS have a polylogarithmic running times for graphs with polylogarithmic maximum degrees. For graphs with arbitrary maximum degree we can use the following algorithm. First, the graph is partitioned into two subgraphs with approximately equal number of nodes, and the subgraphs are recursively colored in Δ+1 colors. Then we iterate through all the colors of one of the subgraphs, recoloring each node with a color different from the colors of all of its neighbors. This algorithm colors a graph with a maximum degree of Δ with $\Delta+1$ colors in $O(\Delta \log \Delta \log n)$ time.

The above algorithms can be implemented in a distributed model of computation [11, 46]. In this model the processors have fixed connections determined by the input graph; processors communicate by sending asynchronous messages over the links, where a processor can send a message to all its neighbors in 1 unit of time. The algorithms in the distributed model achieve the same $O(\log^* n)$ bound as in the EREW PRAM model. It was recently shown that $\Omega(\log^* n)$ time is required in the distributed model to find a maximal independent set on a chain [13, 91]. Our algorithms are therefore optimal (to within a constant factor) in the distributed model.

1.5 Coloring and Matching in Planar Graphs

Euler's formula implies that for every graph of genus γ with n nodes and m edges, $m \leq 3n - 6 + 2\gamma$, and therefore every planar graph has a constant fraction of nodes of degree 6 or less [68]. In this section we use this property in conjunction with the techniques developed above to construct efficient algorithms for coloring and finding maximal matchings in planar graphs.

First we present the algorithm 7-COLOR-PLANAR-GRAPH which finds a 7-coloring of a planar graph in $O(\log n)$ time. The algorithm is shown in Figure 1.3. The first stage of the algorithm partitions the nodes of the graph into sets V_i, such that the degree of any node $v \in V_i$ in $G[V_i + V_{i+1} + V_{i+2}, \ldots]$ is at most 6 (V_i consists of all nodes of degree at most 6 in $G[V - (V_0 \cup V_1 \cup \ldots \cup V_{i-1})]$). Then, the algorithm colors all the subgraphs induced by the node-sets $\{V_i\}$. These graphs are node-disjoint and therefore the coloring can be done in parallel. The last stage of the algorithm adds the subgraphs back in reverse order, updating the coloring.

```
Procedure 7-Color-Planar-Graph
    $V' \leftarrow V$;
    $V_1, V_2, \ldots V_{5\lceil \log n \rceil} \leftarrow \emptyset$;
    $i \leftarrow 0$;
    while $V' \neq \emptyset$ for all $v \in V'$ do in parallel                ⟨⟨ first stage ⟩⟩
        if Degree($v$) $\leq 6$
          then begin
                $V_i \leftarrow V_i + v$;
                $V' \leftarrow V' - v$;
          end;
        $i \leftarrow i + 1$;
    end;
    $k \leftarrow i - 1$;
    for  all  $0 \leq i \leq k$ do in parallel                          ⟨⟨ color the pseudoforests⟩⟩
        $E_i \leftarrow \{\{v, w\} \mid v, w \in V_i \; ; \; \{v, w\} \in E\}$;
        $C_i \leftarrow$ Color-Constant-Degree-Graph($V_i, E_i$);
    end;
    for  $i \leftarrow k$ down to 0 do                                    ⟨⟨ second stage ⟩⟩
        $V'' \leftarrow V_i$;
        for $j \leftarrow 0$ to 6 do
            for  all $v \in V''$ do in parallel
                if $C_v = j$
                  then begin
                        $C_v \leftarrow \max\{\{0, \ldots, 6\} - \{C_w \mid w \in V'; \{v, w\} \in E\}\}$;
                        $V'' \leftarrow V'' - v$;
                        $V' \leftarrow V' + v$;
                  end;
            end;
        end;
    end;
end.
```

Figure 1.3: The 7-Coloring Algorithm For Planar Graphs

Theorem 1.5.1 The algorithm 7-Color-Planar-Graph constructs a valid 7-coloring using n processors and $O(\log n)$ time on a CRCW PRAM.

Proof: By Euler's formula, at least a constant fraction of any planar graph's nodes are of degree 6 or less. Therefore, the first stage partitions the nodes of G into at most $O(\log n)$ sets V_i. We use concurrent reads and writes to determine whether the degree of a node is at most 6, and hence each iteration of the first stage is done in constant time. By Theorem 1.4.1, the second stage uses only $O(\log^* n)$ time. In the ith iteration of the third stage, the graph $G[V_i]$ is already 7-colored and the maximum degree of each node in V_i in the graph $G[V_i + V_{i+1} + V_{i+2} + \ldots]$ is at most 6. Only constant time is then needed to add in V_i and produce a valid 7-coloring of $G[V_i + V_{i+1} + V_{i+2} + \ldots]$. Therefore, only $O(\log n)$ time is used in all three stages. ▌

Remark: If at the first stage, instead of removing from the graph all the nodes with degree of at most 6, we remove all nodes with degree of at most c times average degree (for $c > 1$), the algorithm described above runs in polylogarithmic time for any graph G such that the average degree of any node-induced subgraph G' of G is polylogarithmic in the size of G'. This class contains many important subclasses including graphs that are unions of a polylogarithmic number of planar graphs (*i.e.*, graphs with polylogarithmic thickness [68]).

Given a valid 7-coloring of a planar graph, we can find an MIS in the graph by iterating through colors as in our Constant-Degree-MIS algorithm. With concurrent reads and writes, only constant time is needed for each color class. Hence, we can find an MIS in a planar graph in $O(\log n)$ time on a CRCW PRAM using a linear number of processors.

Remark: Using Euler's formula, we can extend our algorithms for 7-coloring and MIS in planar graphs to graphs of bounded genus γ. We apply the algorithm 7-Color-Planar-Graph as before when there are at least $c\gamma$ nodes remaining in the residual graph, for some constant c. The Heawood map-coloring theorem states that any graph can be colored with $O(\sqrt{\gamma})$ colors, and its proof implies a polynomial time algorithm for finding such a coloring [68]. Therefore, when less than $c\gamma$ nodes remain in the residual graph, we sequentially color it with $O(\sqrt{\gamma})$ colors. With additional time that is polynomial in γ, we can then $O(\sqrt{\gamma})$-color the graph using the same time and number of processors as for 7-coloring a planar graph. Note, that the above algorithm does not need embedding. The related result for MIS on bounded-genus graphs follows as before.

The bottleneck in terms of time and processor bounds in the 5-coloring planar graph algorithms of Boyar and Karloff [22] and Naor [103] lies in $O(\log n)$ applications of a maximal independent set subroutine. This leads to $O(\log^3 n)$ running time using $O(n^3)$ processors by applying Luby's MIS algorithm [93], or alternatively to $O(\log^5 n)$ time using $O(n)$ processors by applying Goldberg and Spencer's MIS algorithm. In [50] we show how to use the Constant-Degree-MIS algorithm described in the previous section to achieve linear-processor $O(\log^3 n \log^* n)$-time algorithm for 5-coloring planar graphs.

Here we present a different approach, which results in a linear-processor algorithm with significantly better running time bounds. The 5-coloring algorithm that is sketched below is essentially a parallelization of the sequential algorithms of Chiba-Nishizeki-Saito [26] and Matula-Shiloach-Tarjan [98]. Given an embedding (which can be computed in $O(\log^2 n)$ time [82]), our algorithm runs in $O(\log n \log^* n)$ time on a CRCW PRAM using a linear number of processors. Given a graph $G = (V, E)$, the algorithm finds a special large independent set I of nodes in G, merges some of the neighbors of I (as described below) and removes the nodes in I to create a new graph G', recursively colors G', and uses this coloring to color the nodes in G.

The special independent set I is constructed as follows. Let Q be the set of all nodes in G of degree greater than 42. Let V_4 be the set of all nodes of degree 4 or less. Let V_5 and V_6 be the set of all nodes of degree 5 with at most one neighbor in Q and the set of all nodes of degree 6 with no neighbors in Q, respectively. Let $S = V_4 \cup V_5 \cup V_6$. Let $G^s = (S, E^s)$ be the graph in which there is an edge between two nodes only if these nodes are connected or have a joint neighbor in $G[V - Q]$, where $G[X]$ denotes the graph induced by the nodes in X. The set I is a maximal independent set in the graph G^s. Since $G[V - Q]$ has constant degree, we can find I using the procedure Constant-Degree-MIS.

In order to construct the graph G', the algorithm proceeds as follows. Start with $G' = G$. For each node in $I \cap V_5$ we find two of its independent (non-adjacent to each other) neighbors that have low degree (42 or less), and merge them into a single supernode. For each node in $I \cap V_6$ we either merge three of its independent neighbors into a single supernode, or merge two of its independent neighbors into two supernodes. The embedding information is used as in [26, 98] to find the neighbors that can be merged while preserving planarity after all nodes in I are removed. Then we remove all the nodes in I to get the graph G'.

After recursively 5-coloring the graph G', we obtain the coloring of G as follows. First color all the nodes of G that correspond to nodes or supernodes of G' with the same color they were colored in G'. Now add all the nodes in I and in parallel color every one of them with a color different from the colors of its neighbors.

In order to bound the running time of the 5-coloring algorithm we need the following lemma, which is similar to Lemma 3 in [26].

Lemma 1.5.2 The size of $S = V_4 \cup V_5 \cup V_6$ is at least a constant fraction of the total number of nodes in the graph.

Proof: Let $R = V - S$. Denote by s_i and r_i the number of nodes of degree i in the sets S and R, respectively. Let $r_* = \sum_{i=7}^{42} r_i$, and let $r_Q = \sum_{i=43}^{\infty} r_i$. By Euler's formula, $r_Q \leq \frac{6}{43}n$.

We prove the lemma by a counting argument. Definitions of r_5 and r_6 imply that $2r_5 + r_6 \leq \sum_{i=43}^{\infty} ir_i$. Euler's formula implies that $3n \geq m$, therefore

$$\begin{aligned}
6n &\geq \sum_{i=1}^{6} is_i + 5r_5 + 6r_6 + \sum_{i=7}^{42} ir_i + \sum_{i=43}^{\infty} ir_i \\
&\geq \sum_{i=1}^{6} is_i + 7r_5 + 7r_6 + \sum_{i=7}^{42} ir_i \\
&\geq 7r_5 + 7r_6 + 7\sum_{i=7}^{42} r_i \\
&\geq 7(\sum_{i=1}^{6} s_i + r_5 + r_6 + r_* + r_Q) - 7\sum_{i=1}^{6} s_i - 7r_Q \\
&\geq 7n - 7|S| - 7 \cdot \frac{6}{43}n
\end{aligned}$$

Thus $|S| \geq \frac{n}{301}$. ▮

Theorem 1.5.3 Given an embedded planar graph, we can color it with 5-colors using n processors in $O(\log^* n \log n)$ time on a CRCW PRAM, and $O((\log^* n + \log \Delta) \log n)$ time on an EREW PRAM.

Proof: Correctness of the algorithm follows from [26] and from the fact that the nodes in I are independent in G^s.

Lemma 1.5.2 implies that the size of S is $\Omega(n)$. The graph G^s has a constant maximum degree and hence the size of the set I is $\Omega(n)$ as well. Therefore the depth of recursion is at most $O(\log n)$.

On a CRCW PRAM, we can find S and Q in constant time as in the algorithm 7-Color-Planar-Graph. The construction of G^s takes constant time because nodes in S have constant degree. The algorithm Constant-Degree-MIS finds I in $O(\log^* n)$ time. In constant time nodes in I can merge appropriate neighbors and delete themselves from G to form G'. Edge lists in G' need not be compacted when we are using the CRCW PRAM. After recursively coloring G', we can color G in constant time.

On the EREW PRAM, $O(\log \Delta)$ additional time per recursion level is needed since we must compact edge lists of G' (so that the set S in G' can be found in constant time). ∎

Remark: Chrobak, Diks, and Hagerup [29] have recently improved the result of Theorem 1.5.3 by giving an algorithm for 5-coloring planar graphs that runs in $O(\log^* n \log n)$ time on an EREW PRAM and does not need an embedding.

Using the techniques described in this section, we can construct a fast algorithm for finding a maximal matching (MM) in a planar (or a constant-degree) graph.

Theorem 1.5.4 A maximal matching in a planar graph can be found in $O(\log n)$ time on a CRCW PRAM using a linear number of processors.

Proof: As in the 7-coloring algorithm, the first stage of the MM algorithm separates the nodes of the graph into sets V_i, such that the degree of any node $v \in V_i$ in $G[V_i + V_{i+1} + V_{i+2}, \ldots]$ is at most 6. Then the graphs $\{G[V_i]\}$ are colored in parallel. The second stage of the algorithm recursively finds MM in the graph $G[V - V_1]$ and removes the matched nodes to get $G[V']$, where V' is the set of the unmatched nodes. The graph $G[V']$ has no edges and the nodes V_1 in the graph $G[V' + V_1]$ have maximum degree of 6. Hence, in 7 iterations over the colors of $G[V_1]$ we can find the MM of G.

1.6 Lower Bounds

In this section we prove two lower bounds for a CRCW PRAM with a polynomial number of processors:

- Finding a maximal independent set in a general graph takes $\Omega(\log n/\log\log n)$ time.
- 2-coloring a directed list takes $\Omega(\log n/\log\log n)$ time.

The first lower bound complements the $O(\log n)$ CRCW PRAM upper bound for the MIS problem that is achieved by Luby's randomized algorithm [93]. The second lower bound complements Theorem 1.3.2. Recall that MAJORITY and PARITY problems are defined as follows: n bits are given in the first n cells of memory. The answer of MAJORITY is 1 if and only if the number of bits that equal 1 is greater than the number of bits that are zero; PARITY is 1 if and only if the number of bits that equal 1 is odd.

Theorem 1.6.1 The running time of any MIS algorithm on a CRCW PRAM with a polynomial number of processors is $\Omega(\log n/\log\log n)$.

Proof: Given an instance of MAJORITY, we construct an instance of MIS in constant CRCW PRAM time. MAJORITY is at least as hard as PARITY [44], and Beame and Hastad have proved that PARITY takes $\Omega(\log n/\log\log n)$ time on a CRCW PRAM with arbitrary instruction set [16, 17]. Therefore the lower bound claimed in the theorem follows.

Let $x_1, x_2, \ldots, x_n$ be an instance of MAJORITY. We construct a complete bipartite graph $G = (V, E)$ with nodes corresponding to '0' bits of the input on one side and nodes corresponding to '1' bits on the other side:

$$\begin{aligned} V &= \{1, \ldots, n\} \\ E &= \{(i,j) \mid x_i \neq x_j\} \end{aligned}$$

To construct this graph, assign a processor P_{ij} for each pair (i, j), where $1 \leq i < j \leq n$. Then, each processor P_{ij} writes 1 into location M_{ij} if $x_i \neq x_j$ and writes 0 otherwise.

A maximal matching in a complete bipartite graph is also a maximum one. By constructing a maximal independent set in the line-graph G' of G, one can find a maximal matching in G.

To construct the graph G', assign a processor P_{ijk} for each distinct $i, j, k \leq n$. Each P_{ijk} writes 1 into location $M_{(i,j),(j,k)}$ if $M_{ij} = M_{jk} = 1$ and 0 otherwise.

The MAJORITY equals to 1 if and only if there is an unmatched node $i \in G$ such that $x_i = 1$, which can be checked on a CRCW PRAM in constant time. ▮

Theorem 1.6.2 **The time to 2-color a directed list on a CRCW PRAM with a polynomial number of processors is $\Omega(\log n / \log\log n)$.**

Proof: We show a constant time reduction from PARITY to the 2-coloring of a directed list. First, we show how to construct, in constant time, a directed list with elements corresponding to all the input bits x_i with value of 1. Let $x_1, x_2, \ldots, x_n$ be an instance of PARITY. We can assume without loss of generality that $x_1 = 1$. With each input cell M_i (that initially holds the value of x_i), associate a processor P_i, a set of processors P_i^{jk} with each index i, $1 \leq k \leq j < i$, a set of cells M_i^j, $0 \leq j < i$, and a cell M_i^l. Initialize all cells that do not store input bits to 0.

In one step, each processor P_i^{jk} reads the value of M_{i-k} and, if it equals to 1, writes 1 into M_i^j, effectively computing the OR-function on the input values $x_{i-j}, x_{i-j+1}, \ldots, x_{i-1}$. Assign a processor P_i^j to each M_i^j, $1 \leq j < i$. Each processor P_i^j reads M_i^j and M_i^{j+1} and writes $(i - j)$ into M_i^l if and only if $M_i^j \neq M_i^{j-1}$. It can be seen that for all $1 \leq i \leq n$, M_i^l holds $\max\{j \mid j < i, x_j = 1\}$.

We have constructed a directed list with elements corresponding to all the input bits x_i with value of 1. Assume this list is 2-colored. Then PARITY equals to 1 if and only if both ends of the list are colored with the same color, which can be checked in constant time. ▮

1.7 Conclusions and Open Problems

We have presented a fast technique for breaking symmetry in parallel and have shown how to apply this technique to improve the running times and processor bounds of a number of important parallel algorithms. We believe that the efficiency of this technique, combined with the simplicity of its implementation, makes it an important tool in designing both parallel and distributed algorithms.

The presented results motivate the following open questions.

- We have proved a lower bound for MIS in general graphs. What is the lower bound for MIS in planar graphs ?

- Beame [15] has proposed the following algorithm for coloring rooted trees of constant degree on PRAM. Run the algorithm 3-COLOR-ROOTED-TREE for $O(\log\log^* n)$ steps. Next, each processor collects the colors of all the descendants on distance $O(\log^* n)$ or less and uses this information and a precomputed lookup table (of size $O(\log^* n \log\log^* n)$) to compute its final color. Given an $\Omega(\log^* n)$ preprocessing time, we can precompute the lookup table; after this preprocessing step, the time to 3-color a tree (or a pseudoforest) will be $O(\log\log^* n)$. Is it possible to 3-color a tree in $o(\log^* n)$ time on PRAM with no preprocessing?

- Can we compute an MIS in general graphs in $o(\log n)$ time ?

- Our $\Delta+1$-coloring algorithm implies that $\Delta+1$ coloring is a "local" property for constant-degree graphs. In other words, if each node in a distributed system will gather all the ID's from nodes that are at distance of at most $O(\log^* n)$ from it, it will be able to compute its color. Linial has shown that Δ^2 coloring is a local property for general graphs [91]. A natural question to ask is whether $\Delta + 1$-coloring is a local property for general graphs.

- Direct implementation of Luby's deterministic MIS algorithm [93] in a distributed network leads to superlinear running time, because after each iteration we have to decide which one of the n^2 executions is the best. Is it possible to find an MIS in a general distributed network deterministically in sublinear time ?

Chapter 2

Sticky Bits and Universality of Consensus

2.1 Introduction

The usual way to write programs for shared memory multiprocessors is to use synchronization primitives, that allow mutual exclusion. The main problem with this approach is that it causes one processor to wait for another, essentially reducing the speed of the system to the speed of the slowest component, which can be zero if this component has failed.

In his seminal paper, Lamport suggested that usually we do not need to ensure that the data is really accessed in a sequential fashion [84]. Instead, we need a mechanism that makes the accessing processors see *as if* the data is accessed sequentially. Lamport defined the notion of *atomicity*, where intuitively an object is atomic if from the point of view of accessing processors the accesses do not overlap in time.

Atomicity is trivially achieved using locking, at the expense of causing unnecessary waiting of one processor for another. Hence, the question is whether we can construct atomic objects that are also wait free. Lamport has suggested studying the special case of constructing wait-free atomic registers from safe ones without using mutual exclusion [84]. (Informally, "safe" means that the register operates correctly as long as there are no accesses that overlap in time.) This problem has gained popularity, and has led to a large number of papers, that describe constructions of various atomic registers [20, 104, 109, 110, 128].

Using safe registers as our basic primitive objects has several inherent disadvantages. First, the constructions based on safe registers are so complex and so hard to verify that even some of the published ones are erroneous (see Schaffer [114] for a more detailed discussion). On the other hand, Dolev, Dvork, and Stockmeyer [35], Chor, Israeli, and Li [27], and Loui and Abu-Amara [92], have proved that safe registers are not sufficient in order to reach even 2-processor wait-free consensus. From this Herlihy [69] has concluded that safe registers are not sufficient to implement wait-free versions of simple data objects like queues, stacks, etc.

Thus, we are lead to the conclusion that a new primitive object should be defined. This object should be powerful enough to be used instead of mutual exclusion, it should provide a convenient level of abstraction for writing concurrent programs, and it should be primitive enough to be easily implemented. In this chapter we define such object, the Sticky Bit, which is a special case of a 3-valued memory cell that supports a restricted variant of an atomic Read-Modify-Write.

The Sticky Bit can be viewed as a generalization of consensus, which has proven to be a valuable tool in understanding the limitations of asynchronous distributed systems [35, 39]. On the other hand, the definition of the Sticky Bit is memory-oriented, which makes it a convenient alternative to consensus in the context of shared-memory systems. In particular, Sticky Bit objects allow us to easily write algorithms where several processors "help" each other so that the task is completed with the speed of the fastest processor.

We illustrate the importance of the "help" paradigm by presenting a very simple wait-free leader-election algorithm. It should be noted that this algorithm is interesting by itself, because the standard methods of Coan and Turpin [30] that convert a single-bit consensus into a multibit one do not seem to be applicable in the context of wait-free computation. Employing the "help" paradigm we prove that Sticky Bit is "universal" by showing how to use $O(n^2 \log n)$ Sticky Bits to transform any safe implementation of a sequential object into an atomic one, where n is the number of participating processors.

A Sticky Bit object can be easily constructed from two safe bits and a single initializable object that implements a wait-free single-bit consensus, where "initializable" means that the object can be initialized by one of the participating processors so that it can be used again to reach consensus, as long as the initialization does not overlap any other operation. The construction presented in this chapter indicates that reaching consensus is the fundamental problem in wait-free synchronization. In particular, randomized consensus algorithms of Chor, Israeli, and Li [27] and Abrahamson [1] together with our construction imply that polynomial number of safe bits is sufficient to convert a safe implementation of a sequential object into a atomic (randomized) wait-free implementation. The only other attempt to design such a construction was done by Herlihy [69], whose construction requires an unbounded number of additional safe bits and assumes that the model supports powerful memory-management operations.

The chapter is organized as follows. Section 2.2 presents a formal model of a shared-memory multiprocessor. The formal definitions introduced in this section are used in Section 2.3 to define the notions of wait-freeness and atomicity. Section 2.4 presents definition of the Sticky-Bit object and illustrates its use by presenting a wait-free leader election algorithm. In Sections 2.5 and 2.6 we describe how to construct an atomic simulation of any sequential object from Sticky-Bits, which proves that Sticky-Bit data object is universal. Conclusions and open problems are presented in Section 2.7.

2.2 Model

Informally, we regard a shared-memory multiprocessor executing a number of sequential threads as a set of asynchronous processes communicating through shared data objects, where each process corresponds to an execution of a procedure. Each sequential thread is executing a single procedure at a time, where execution of the "call" instruction causes it to suspend the current procedure and start executing the one that was invoked. The suspended procedure is not continued until the invoked one executes the "return" instruction. This corresponds to a system that does not support instructions that create new processes, *i.e.* all threads exist from the beginning.

We use I/O Automata of Lynch and Tuttle [96] with the addition of ports, as described in [95]. We view all communication as if it is done through *I/O channels*, where each channel has two endpoints called *ports*. For each channel, one port is called the *master* and another one is called the *slave*, where slave ports correspond to entry-points of procedures and master ports correspond to "call" instructions. Messages sent from master ports are called *commands* and messages sent from slave ports are called *responses*.

Processors and data objects are modeled as non deterministic automata with possibly a countably infinite number of configurations and possibly a countably infinite fan-out from every configuration. An Input/Output Automaton is a tuple $M = (\mathcal{E}, Q, Q^0)$, where $\mathcal{E} = \mathcal{E}^{out} \cup \mathcal{E}^{in} \cup \mathcal{E}^{int}$ is the set of actions ($\mathcal{E}^{out}, \mathcal{E}^{in}, \mathcal{E}^{int}$ are output, input, and internal actions, respectively), Q is the set of states, and $Q^0 \subseteq Q$ is a distinguished set of start states. Each action corresponds to a transition of the automaton from one state to another; we say that the actions which correspond to transitions out of a given state are *enabled* in this state. An execution of an automaton is represented by a sequence of actions, which we call a *schedule*.

A *port automaton* is a tuple (Val, Π, CH, M), where

- *Val* is the set of values that can be sent as messages.
- Π is the set of ports. Each port has a *type*, which is either *master* or *slave*.
- CH is the set of channels. Each channel is a pair of ports, one of type master and the other one of type slave. A port can belong to at most one channel; a port which does not belong to a channel is called *external*, and the rest are called *internal*.
- M is an input/output automaton with the property that every action is a tuple (m, π); $m \in Val$, $\pi \in \Pi$. We consider only schedules where any action (m, π_1) is followed by (m, π_2) if $ch = (\pi_1, \pi_2) \in CH$. To simplify notation, we write (m, ch) in this case. All external actions are associated with external ports; all internal actions are associated with internal ports.

An output action associated with a master port or an input action associated with a slave port is called a *command*; an input action associated with a master port or an output action associated with a slave port is called a *response*. A port automaton is *well formed* if any schedule H restricted to any port π starts from a command and consists of alternating commands and responses. We say that port π is *input enabled* in a state C if there exists an input action associated with this port, that is enabled in C. A schedule H is *balanced* if it brings the system to a state in which a port is input-enabled if and only if it is of type slave.

We say that an automaton has a *procedure signature* if it has at most one slave port; an automaton has an *object signature* if it has no master ports. In order to abbreviate, we refer to these automata as procedures and objects, respectively.

A composition of several port automata with disjoint set of ports is done exactly like the composition of input/output automata. In order to model interactions between components we can link one component to another by defining new channels. A composition of a number of port automata with additional channels is called a *system.*

An object is specified by describing its external ports and stating all of its legal external schedules. We say that an automaton implements an object O if it has the same set of external ports and its external schedules are a subset of the schedules specifying O.

It is important to formalize the notion of *implementing* one object *in terms of another*. An implementation of an object O in terms of objects $O_1, O_2, \ldots, O_k$ is a port automaton that implements O and is a system constructed by interconnecting a number of port automata such that each automaton in the system is either a procedure or it implements one of the objects $O_1, O_2, \ldots, O_k$.

An example of an implementation of one object in terms of other objects is given in Figure 2.1. Arrows correspond to channels and are directed from master to slave ports; circles and ovals correspond to procedure and object automata, respectively. Consider the directed graph with nodes corresponding to the automata and the edges corresponding to channels. Observe that this graph is acyclic. Associate with every external port π_i the system that consists of the interconnection of all automata with procedure signatures that correspond to nodes reachable from the node that corresponds to the automaton that owns π_i, and denote the obtained automaton by $\mathcal{P}_i$. Note that $\mathcal{P}_i$ has a procedure signature. Intuitively, this corresponds to decomposing the system into "front-ends" and "representation objects", and we call this *canonical decomposition.*

With each system $\mathcal{A}$ we associate a system $\tilde{\mathcal{A}}$ that hides the actions of $\mathcal{A}$ that correspond to internal channels. For each schedule H of $\mathcal{A}$, the corresponding schedule of $\tilde{\mathcal{A}}$ is called

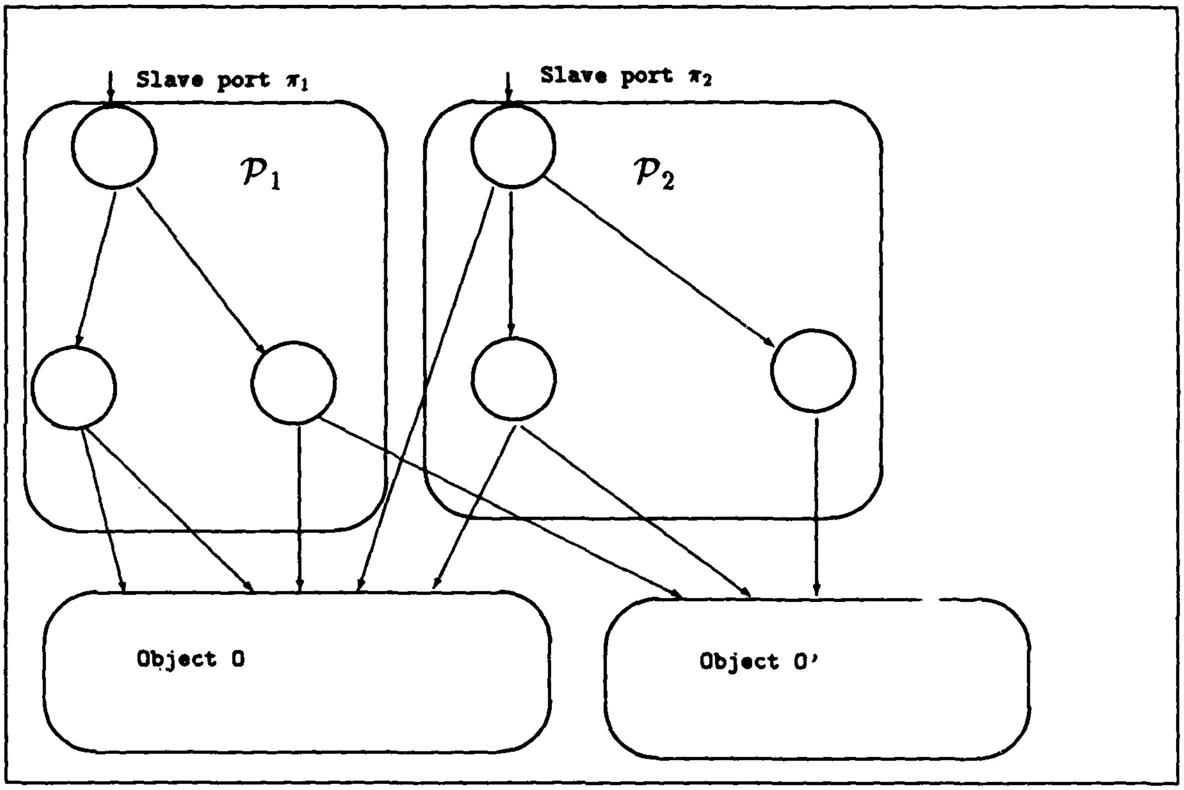

Figure 2.1: Implementation of an object in terms of objects O and O'. Arrows correspond to channels and are directed from master to slave ports; circles and ovals correspond to procedure and object automata, respectively.

the *external schedule* and is denoted by $H|\tilde{A}$. The actions in $H|\tilde{A}$ are divided into actions corresponding to the slave ports of $\tilde{A}$, and actions corresponding to the master ports of $\tilde{A}$, which we denote by $H|\tilde{A}^s$ and by $H|\tilde{A}^m$, respectively.

2.3 Wait-Free Atomicity

Two properties of an atomic register that make it an interesting data object are the ability to perform correctly when the reads and writes are asynchronous and concurrent, and the ability to withstand fail-stop "death" of one of the processors that access it, even if this death is in the middle of an access. In this section we define these properties precisely using the formalism introduced in the previous section, and generalize them to apply to more complicated objects.

It is natural to give a specification of a data object in terms of the behavior of this object when every command sent to the object is executed and acknowledged before a new command is issued. For example, in the case of a register, this corresponds to executions in which reads or writes do not overlap. The following definition, due to Herlihy and Wing [70], captures this idea.

Definition 2.3.1 A schedule S is *sequential* if every *command* action in S is immediately followed by the corresponding *response* action on the same port. A schedule S is *sequential* with respect to component O, if $S|\tilde{O}^s$ is sequential. An object is sequential if any schedule which is sequential w.r.t. this object can be extended to a balanced schedule which is also sequential w.r.t. this object.

Objects like registers, queues, stacks, etc. are examples of sequential objects (if we assume that a "dequeue" operation on an empty queue or "pop" of an empty stack are defined to return exceptions). On the other hand, consider a 2-port object that, given a value on one of the ports, responds with this value or with the value sent to it through the other port, whichever is larger. This object is not sequential because the response to the first command has to come after the second command, *i.e.* the waiting is inherent in the specification of the object.

Lamport [84] defined a register as *safe* if a "read not concurrent with any write obtains the most recently written value". Generalizing to arbitrary sequential objects, we have the following definition.

Definition 2.3.2 An implementation of a sequential object is safe if the set of all schedules which are sequential w.r.t. this object is a subset of the sequential schedules specifying the object.

What is the natural way of specifying the behavior of a sequential object for non-sequential schedules ? When writing programs that use sequential objects, we usually disregard the possibility of overlap and use the object as if we are assured that the only possible schedule will be sequential with respect to this object. In other word, we assume that the object responds immediately to any request. Intuitively, the specification of what is the "correct" behavior of a sequential object when a number of accesses overlap has to take this into account, so that a program that is correct under the assumption that the schedule is sequential with respect to the object, *will also be correct for non-sequential* schedules.

For the case of registers, Lamport has defined the notion of *atomicity*. Intuitively, an object is atomic if it behaves as if each operation occurs somewhere between the command and the response. In order to generalize the notion of atomicity to arbitrary sequential objects, we first define a partial order on *operations* in a given schedule H, where an operation is a command action followed by a response action in the restriction of H to the external ports of the object. Let $o = (e_c, e_r)$ and $o' = (e'_c, e'_r)$ be two operations in H. Recall that e_c, e'_c are commands and e_r, e'_r are responses. Then $o \prec_H o'$ if both e_c and e_r appear before e'_c in H.

Definition 2.3.3 [70] An object is *atomic* if for every external schedule H of this object, there exist schedules H' and S, such that H' is a balanced extension of H, S is sequential schedule consisting of the same actions as H', and $\prec_{H'} \subseteq \prec_S$. The schedule S is called linearization of H and is denoted by $\mathcal{L}(H)$.

Translating Lamport's notion of the *global-time* model to our definitions, a global-time model of a schedule H is an assignment of a real number $T(e)$ to each action e in H, such that if $H = H_1 e_1 H_2 e_2 H_3$, then $T(e_1) < T(e_2)$. In other words, later actions are associated with larger numbers. Denote by $\mu_H(o)$ the interval associated with operation o. Then $o_1 \prec_H o_2$ if and only if the interval corresponding to $\mu_H(o_1)$ lies completely to the left of $\mu_H(o_2)$. Therefore, if two schedules H and H' consist of the same operations and for each operation o we have $\mu_{H'}(o) \subseteq \mu_H(o)$, then $\prec_H \subseteq \prec_{H'}$. On the other hand, Proposition 1 in [84] states that if $\prec_H \subseteq \prec_{H'}$, then there exists $\mu_{H'}$, such that for each operation o we have $\mu_{H'}(o) \subseteq \mu_H(o)$. Thus, the above definition of atomicity can be reinterpreted as saying that any external schedule should correspond to some sequential schedule consisting of the same operations, such that each operation of the sequential schedule occurs within the interval of time corresponding to the operation of the given schedule.

The following lemma shows that the definition of atomicity confirms to the intuition described above. Intuitively, this lemma claims that if an object is atomic, then in order to prove the correctness of the system, it is sufficient to prove correctness under the assumption that accesses to the object do not overlap. The proof of this lemma is similar to the proof of Lemma 3.2 in Herlihy and Wing [70].

Lemma 2.3.4 Consider a system A with decomposition $(P_1, P_2, \ldots, P_k, O_1, O_2, \ldots, O_l)$, such that for any schedule H, the schedules $H|\tilde{P}_i^m$ are sequential for all P_i. Then the set of all possible external schedules of A is the same as the set of external schedules of A which is obtained from schedules H such that $H|\tilde{O}$ is sequential, where $O = (O_1, O_2, \ldots, O_l)$.

Proof: Consider schedule H of the system A. Let schedule $H|\tilde{O}$ be the external schedule of O. Consider an object O_i and its corresponding external schedule $H|\tilde{O}_i$. The object O_i is atomic, and therefore there exists a linearization $\mathcal{L}(H|\tilde{O}_i)$, that imposes complete order $\prec_{o_i}$ on the operations in $H|\tilde{O}_i$. On the other hand, we assume that each procedure automata imposes a complete order $\prec_{p_i}$ on operations associated with its master ports. Consider the union of relations $\prec_{o_i}$ and $\prec_{p_i}$ for all i. We claim that it is a partial order, and therefore can be extended to a complete order. If this is true, there exists a schedule S that is sequential with respect to O, and $S|P_i = H|P_i$, and therefore the processors can not distinguish between S and H and hence produce correct responses.

Assume that we do not have a partial order, which means that we have a cycle. Objects $O_1, O_2, \ldots, O_l$ can not interact among themselves directly, because an object has slave channels only. The only possible interaction is through some processor that is connected to a number of objects. Therefore, if all operations on the cycle are related by $\prec_{o_i}$ for some i, they are related by $\prec_{o_i}$ for the same i, which is impossible because $\prec_{o_i}$ is a total order for each i.

Hence, at least two operations ω_1 and ω_2 that are consecutive on the cycle, are related by a member of $\prec_{p_i}$ for some i, that is $\omega_1 \prec_{p_i} \omega_2$. Denote the union of partial orders $\prec_{p_i}$ and $\prec_{o_i}$ by $\prec_u$. The existence of cycle means that $\omega_2 \prec_u \omega_1$. The schedule is sequential with respect to master ports of P_i, and therefore $\omega_1 \prec_{p_i} \omega_2$ means that $\omega_1 \prec_H \omega_2$. But the partial order $\prec_u$ is consistent with H, which means that $\omega_2 \not\prec_u \omega_1$. Thus, existence of a cycle in $\prec_u$ leads to a contradiction. ▮

Atomicity is only one of the properties required by Lamport of "atomic registers". Another, not less important property, is "wait-freeness". Intuitively, the idea is that the time it takes to access an object should depend only on the speed of the accessing processor. In particular, the object has to eventually return a response, even if all the rest of the processors "died" in the middle of an operation. Note that without this requirement atomic objects can be trivially constructed using busy-waiting.

Formally, wait-freeness is defined as follows. Consider a system with canonical decomposition $(P_1, P_2, \ldots, P_k, O_1, O_2, \ldots, O_l)$ where external slave port π_i corresponds to component P_i. Let $\mathcal{H}(C)$ denote the set of possible schedules that start at state C.

Definition 2.3.5 Let $\mathcal{H}_N^{p_i}(C) = \{H : H \in \mathcal{H}(C), \|H|(P_i, O_1, O_2, \ldots, O_l)\| > N\}$. The signature of an external slave port π_i is *wait-free* if there exists N, such that:

1. For any state C in which the port π_i is not input-enabled, there exists a schedule $H \in \mathcal{H}_N^{p_i}(C)$ such that it does not include any commands associated with external ports of $(P_i, O_1, O_2, \ldots, O_l)$.

2. For any state C in which the port π_i is not input-enabled, and for any schedule $H \in \mathcal{H}_N^{p_i}(C)$, there is a prefix H' of H, s.t. π_i is input-enabled in state $H'(C)$.

We say that a system is wait-free if and only if all external slave ports of the system have wait-free signatures. Note that the object O does not have to be wait-free in order for the system $(P_1, P_2, \ldots, P_k, O)$ to be wait-free. It should be noted that our definition is different from the one given in [69], which fails to take into account the possibility of "starvation" of slow processors. In other words, there is an implicit "fairness" in our definition, because the constant N does not depend on what is happening on other ports. For example, the above definition has the property that if two processors access a shared queue, in order for this queue to be considered wait-free it must "reply" to the slower processor even if the faster one is infinitely faster.

After formalizing the notions of atomicity and wait-freeness, we can define what it means for an atomic object to be *universal*.

Definition 2.3.6 A data object O is *universal* if given a *safe* implementation of a sequential object O', we can construct a wait-free system that atomically implements O' in terms of O-type objects.

2.4 Sticky Bit

In this section we introduce a new data object, the *Sticky Bit*, and illustrate the use of this object by presenting a deterministic wait-free leader-election algorithm.

Definition 2.4.1 An atomic *Sticky Bit* (ASB) is a data object that holds values in $\{\perp, 0, 1\}$ and atomically supports the following operations:

- JAM(v) – If the value was $\perp$ or v, sets it to v and returns "success". Otherwise returns "fail".
- READ – Returns the current value of the object.

In addition, it supports a non-atomic FLUSH operation.

- FLUSH – Sets the value to $\perp$. This operation is non-atomic in the sense that any other operation that overlaps it produces unpredictable results.

A consensus protocol was defined in the seminal paper of Fisher, Lynch, and Paterson [39] to be a protocol where each processor has a 1-bit input and produces a 1-bit output which confirms to two conditions. First, all produced outputs are the same, and second, if the output is v then there is at least one participating processor whose input is v. The *consensus protocol* can be naturally represented as an n-port object. It is easy to see that it is possible to construct an ASB from an initializable single-bit consensus object and two safe bits, where "initializable" means that after using the object to reach consensus it is possible to initialize it and use it again, as long as the initialization does not overlap with any other operation. In particular, this implies that the randomized consensus algorithms of Chor, Israeli, and Li [27] and Abrahamson [1] can be used to construct a randomized wait-free atomic Sticky Bit. Note also that ASB is a special case of the write-once memory, and can be easily implemented in hardware.

The usual problem that arises when designing wait-free algorithms is that even if it is sufficient that a single processor will execute some task, we can not assign a specific processor to this task because the adversary might make this processor fail-stop. ASB objects provide a convenient way to address this problem. In particular, they allow several processor to execute the same task concurrently, "helping" each other, without interfering one with another.

```
Procedure JAM($v_i$)
    $i, i' \leftarrow$ the ID of the processor;
    Jam-0($g_i$);
    for $j \leftarrow 1$ to $l$ do
        $b \leftarrow j$−th bit of $v_{i'}$;
        jam $j$th bit of $v$ with $b$;
        if the jam failed
        then begin
          for $k \leftarrow 1$ to $l$ do
              if $g_k = 0$ and $\forall\, 1 \leq k' \leq j$, ($k'$th bit of $v_k$) = ($k'$th bit of $v$)
              then $i' \leftarrow k$;
          end;
        end;
    end;
    return $v$;
end.
```

Figure 2.2: Code of *Jam*(v_i), executed by processor i.

A simple example that illustrates how to use this capability, is a wait-free atomic implementation of a "Sticky-Byte" object. This object is similar to the Sticky-Bit, but holds a number of bits (say l) instead of a single one. The command that corresponds to JAM(0) or JAM(1) of ASB is JAM(v), where v is an l-bit value. Similarly to ASB, JAM(v) returns "success" if the object holds v after JAM(v) returns, and "fail" otherwise.

Observe, that an implementation that is based on representing a Sticky-Byte by l ASB objects where each processor simply tries to jam its bits one-by-one, leads to incorrect values. For example, consider the case where $l = 2$ and one processor tries to jam $(1, 0)$ and the other one tries to jam $(0, 1)$. A possible scenario is that the first bit is jammed by the first processor and the second bit is jammed by the second processor, leading to the incorrect value of $(1, 1)$. On the other hand, if a processor "returns" immediately after it comes to the conclusion that it must return "fail", then some of the bits of the Sticky Byte might remain undefined if the processor that is supposed to return "success" is stopped by the adversary.

The main idea is to require any processor that recognizes that he must return "fail" *to help* the processor that might still return "success" — a "help thy neighbor" paradigm. The code executed by processor p_i in order to simulate JAM(v_i) operation is shown in Figure 2.2.

The algorithm begins with p_i storing its input into v_i and marking that its v_i is valid by jamming g_i to 0. Then, it executes l iterations, where l is the number of bits in the input. At iteration j, p_i tries to jam the jth bit of the decision to be the jth bit of $v_{i'}$, where initially i' is the processor's ID. If it does not succeed, it means that there exists k and some processor $p_{k'}$, such that $p_{k'}$ has succeeded in jamming the jth bit of v_k into the jth bit of v. By induction on the number of iterations, we see that there exists at least one v_k such that $g_k = 0$ and the first j bits of v_k correspond to the first (already jammed) j bits of v. The processor p_i finds such k, and from now on tries to jam v_k into v, essentially "helping" processor p_k. Note that this algorithm has an interesting property that even if a processor p_i stops immediately after jamming 0 into g_i, its input may still be the decision jammed into v.

Observe, that if each processor tries to jam its own ID, the above algorithm implements a wait-free leader-election in $O(\log n)$ time. This implies that a Sticky Byte that holds an arbitrary number of bits can be implemented from $\log n$ Sticky Bits, where an access time of such implementation is $O(\log n)$.

2.5 Atomic Implementation of an Arbitrary Object.

In order to construct an atomic simulation of an arbitrary sequential object, we must be able to impose an order on accesses that overlap in time, such that this order will be consistent with their real order (see Definition 2.3.3), *i.e.* if an access was completed before another one was started, then the same relation between these accesses should exist in the imposed order. In other words, we must construct a sequential schedule which is consistent with the real schedule.

A natural approach, proposed by Herlihy [69], is to assume that the system supports an atomic operation that prepends an element to the beginning of the list. The idea is that a processor that wants to access the object stores the command in the list, and then uses the commands that were stored beforehand in this list to compute the state of the object and the appropriate response. The commands stored in the list correspond to a sequential schedule which is consistent with the real one, and therefore the implementation is atomic.

In order to be able to show that it is possible to implement such list from Sticky Bit objects, we review the construction of [69], adding several important details. Upon receiving a command *cmd*, processor p_i proceeds as follows:

1. Gets a free cell *Cell* and stores *cmd* in this cell.

2. Uses APPEND to prepend *Cell* to the list.

3. Reads the cells in the list one by one to construct the suffix S of the sequential schedule of the simulated object. The cells are read until it encounters a cell that holds a state instead of a command.

4. Computes the "current" state of the object that results from applying S to the encountered state and stores it in *Cell.*

5. Frees cells of the list that belong to it and that have at least n cells that hold states (and not actions) ahead of them in the list.

6. Computes the response *rsp* of the object and returns it.

Observe, that because of Step 4 there are at most n cells in the list that hold actions and not states. Hence, in order to compute the "current" state of the object it is enough to scan at most n cells of the list, and therefore the implementation is wait-free. Step 5 is needed to bound memory. As a result of the fact that a processor never scans more than n cells of the list and the fact that it stops scanning after encountering a cell that holds a state, Step 5 does not remove from the list any cell that might be read by some processor.

Though it seems that the main problem lies in implementing the APPEND command, there are two important issues which were not considered by Herlihy in [69]. First, note that a straightforward implementation of Step 5, *i.e.* recognizing that there are at least n cells that hold states ahead in the list by scanning these cells, is incorrect. The problem lies in the fact that some of these cells might be already freed, initialized, and used again, causing us to follow "dangling" pointers. One way to solve this problem is to add n bits $\{b_1, b_2, \ldots, b_n\}$ to each cell, *initially all 0*. After a processor modifies its current cell to hold a state, it scans the following n cells in the list, writing 1 in the appropriate bit of each cell, according to the distance to this cell. Each time a processor is invoked and needs a new cell, it first checks all of its cells that are still in the list, and frees those that have all the bits b_i equal to 1. This way the list is traversed only in the forward direction. Moreover, if a processor accesses a cell during this traversal, then the appropriate b_i bit of this cell was 0 before the access, and therefore this cell could not have been initialized, which means that we never follow "dangling" pointers.

The second point concerns the space complexity. Herlihy claims in [69] that because there are at most n cells in the list that hold actions and not states, and because each such cell can delay at most n other cells from being freed from the list, the space complexity is $\Theta(n^2)$ in the worst case. This claim is correct only if there is a single bounded-size pool from which new cells are allocated. Unfortunately, it is impossible to implement such pool from safe registers, because it allows wait-free 2-processor consensus. Therefore, it seems that the actual space complexity of Herlihy's construction (which assumes that the system supports a wait-free atomic operation that prepends an element to the beginning of a list[1]) is $\Theta(n^3)$ in the worst case, which is achieved by allocating disjoint pools to each one of the processors.

The code which includes the details discussed above is presented in Figure 2.3. It assumes that the list object supports the following three atomic operations:

- GFC – Gets a free cell.
- INIT – Frees and initializes the given cell.
- APPEND – Prepends the given cell to the beginning of the list.

2.6 Universality of Sticky Bit

We show universality of the atomic Sticky Bit object by presenting an implementation of the GFC, INIT and APPEND operations. Similarly to the implementation of the Sticky Byte, the heart of the presented algorithms is the idea of extending "help" to less fortunate processors.

In order to implement the operations atomically, the procedures described in this sections "busy-wait" until the operation is executed and only then return. Thus, the problem is to limit the amount of this waiting in order to achieve wait-freeness, and this is where the "help"

[1] Herlihy's construction of a wait-free atomic object directly from multibit consensus uses unbounded memory.

```
procedure UNIVERSAL(cmd)
    ActionList – the list that stores actions and states;
    Tmp(n) –a local array of n cells;
    MyCells – a local set of pointers to processor's own cells in the ActionList;
    Cell,NextCell – local variables, correspond to cells in the lists;
    state,NewState – local variables that can hold states;
    cmd, rsp – input command and output response;

{Release "useless" cells}

    for all $Cell \in MyCells$ do
        if $\forall i,\ 1 \leq i \leq n:\ Cell.b_i = 1$ then begin
            INIT(Cell, ActionList);
            if "success" then $MyCells \leftarrow MyCells - Cell$;
        end;
    end;

{Get a free cell}

    $Cell \leftarrow$ GFC;

{Append the cell to the list}

    $MyCells \leftarrow MyCells + Cell$;
    store command cmd in Cell.Data, initialize all $Cell.b_i$ to 0;
    $NextCell \leftarrow$APPEND(Cell, ActionList);

{Compute the "current" state}

    $i \leftarrow 1$;
    while type(NextCell) = action do
            $Tmp(i) \leftarrow NextCell$;
            $NextCell \leftarrow Cell.Next$;
            $i \leftarrow i + 1$;
    end;
    $state \leftarrow NextCell.Data$;
    apply actions in $Tmp(j),\ 1 \leq j \leq i$ to state to compute NewState;
    apply command cmd to NewState to compute response rsp;
    (NextCell.Data, NextCell.Type)$\leftarrow$(NewState, "state");

{Mark n cells following the appended cell}

    for $i \leftarrow 1$ to $n$ do
        $NextCell.b_i \leftarrow 1$;
        $NextCell \leftarrow NextCell.Next$;
    end;
    return rsp;
end.
```

Figure 2.3: Atomic simulation of an object.

Claimed: Equals to 1 only if the cell is not free. (Sticky)

ProcID: Holds the processor ID that "owns the cell". (Sticky)

NotHead: Equals true when the cell is in the list but is not the head of the list. (Sticky)

Data: The value of the cell. (Safe)

DataValid: Equals to 1 if the *Data* field is properly filled with data. (Sticky)

Next: Points to the next cell in the list. (Sticky)

Prev: Points to the previous cell in the list. (Sticky)

Init: Equals 1 if the cell is being initialized. (Safe)

$r_1, r_2, \ldots, r_n$: The bit r_i equals to 1 if processor p_i does not want the cell to be initialized. (Safe)

CountInit: The highest i such that for all $j \leq i$, the owner of the cell saw each one of r_j being equal to 0. (Safe)

Figure 2.4: Information stored in a single cell

paradigm comes into play. Each implementation of an operation is constructed from two procedures: the kernel procedure which actually implements the operation, and the control procedure, which repeatedly invokes the kernel. The kernel procedure is designed so that it either succeeds in executing the operation or returns "failure". Moreover, if it returns "failure" then at least one other processor has succeeded in executing the same operation concurrently. The control procedure repeatedly invokes the kernel procedure until it returns with success, and then invokes the kernel procedure *on behalf* of each one of the processors that is currently trying to execute this operation. This ensures that there can be at most n consecutive "failures" of kernel procedure, which makes our algorithms wait-free. Sticky Bits are used in order to enable one processor to execute on behalf of the other one without interfering with him; essentially, the idea is that the processor that is being helped does not "know" this.

The information stored in each cell of the list is shown in Figure 2.4. In particular, the *Prev* field is constructed as an atomic Sticky Byte, and is used to decide which processor succeeds in appending his cell to the list; *ProcID* is also a Sticky Byte which is used to decide which processor currently "owns" the cell (this processor is responsible for initializing it).

Implementation of INIT Initializing a cell involves, in particular, flushing the *Prev* field. We use the GRAB and RELEASE procedures, shown in Figure 2.5, to prevent the flushing while there exists a possibility that some processor might read this field, because by definition of the

```
procedure GRAB(Cell);
    if Cell.Init = true
       then return "fail";
       else begin
         $Cell.r_i \leftarrow 1$;
         if Cell.Init = true
            then begin
              $Cell.r_i \leftarrow 0$;
              return "fail";
            end;
            else begin
              if Cell.DataValid = true
                 then return "success";
                 else return "empty";
            end;
       end;
end.

procedure RELEASE(CELL);
    $CELL.r_i \leftarrow 0$;
end.
```

Figure 2.5: The GRAB and RELEASE procedures executed by processor p_i

Sticky Bit, FLUSH can not be overlapped by any other operation. Before accessing a cell, we require that the processor will GRAB it first, by checking the *Init* bit, setting the appropriate r_i, and checking the *Init* bit again.

The initialization of a cell is done by the processor that "owns" it, *i.e.* the processor whose ID is stored in the *ProcID* field of the cell. This processor first sets *Init* bit to state that the cell is under initialization, and then it checks the r_i bits one by one. Note that if $r_i = 0$ for some i after *Init* bit is set to zero, p_i will not be able to GRAB the cell as long as *Init* stays nonzero. Thus, by deferring the initialization until the processor who owns the cell sees each one of the r_i bits being zero at least once, we guarantee that the access to the "sticky" data in the cell and the initialization of the cell are never executed concurrently.

Implementation of GFC The code implementing the GFC operation is presented in Figure 2.7. To get a free cell, the processor p_i first sets $AnnounceGFC(i)$ to state that it is in the middle of trying to get a free cell and scans all memory checking whether there is an empty

```
procedure INIT(Cell);
     if Cell.Init = 0 then Cell.Init ← 1;
     RELEASE(Cell);
     j ←Cell.CountInit;
     while j < n and $r_j = 0$ do begin
        j ← j + 1;
     end;
     Cell.CountInit ← j
     if j = n
       then begin
          initialize the cell;
          Cell.CountInit ← 0;
          Cell.Init ← 0;
          return "success";
       end;
       else return "fail"
end.
```

Figure 2.6: The INIT procedure.

cell that was "prepared" for him, *i.e.* a cell with *Claimed* bit off and *ProcID* being equal to i. If such cell was not found, it scans the cells one-by-one, trying to jam i into the *ProcID* field of the cell; the scan continues until success. When the processor has succeeded, *i.e.* the *ProcID* field of the current cell C holds i, it checks whether C is his "own" cell by checking the C.*Claimed* bit. If C was already claimed, the search continues. Otherwise, it sets C.*Claimed* to "true" and resets *AnnounceGFC*(i). C is the cell that is eventually returned as the response to the GFC command.

Before returning, the processor *helps* all other processors that are in the middle of looking for a free cell. For each such processor p_j, the processor p_i scans all the cells, looking for a cell with j jammed into *ProcID* and the *Claimed* bit equal to 0. If such a cell was found, it means that there is a cell that only p_j can claim, and in this case p_i returns. If not, it starts participating in the search again *on behalf of* p_j. The only difference is that after succeeding with jamming j into *ProcID* of a cell with *Claimed* bit 0, it does not turn on the *Claimed* bit.

Observe, that if during one memory scan no cells were allocated, a free cell is found. The following lemma bounds the number of scans in which a free cell was not found.

Lemma 2.6.1 At most $2n^2$ cells can be allocated from the moment a processor announces that it is trying to get a free cell until a cell with *ProcID* being equal to the ID of this processor is actually

```
procedure GFC;
    AnnounceGFC(i) $\leftarrow$ 1;
    Cell $\leftarrow$ GFC-INNER(i);
    Cell.Claimed $\leftarrow$ true;
    AnnounceGFC(i) $\leftarrow$ 0;
    for all j such that AnnounceGFC(i) = 1 do begin
        Tmp $\leftarrow$ GFC-INNER(j);
        RELEASE(Tmp);
    end;
    return Cell;
end.

procedure GFC-INNER(ID);
    for all cells Cell in memory do begin
        if GRAB(Cell)= "empty" and Cell.ProcID = ID and Cell.Claimed = 0
          then begin
            return Cell;
          end;
    end;
    do  forever
        for all cells Cell in memory do begin
            if GRAB(Cell) = "empty"
              then begin
                try to jam ID into Cell.ProcID;
                if "success" and Cell.Claimed = false
                  then return Cell;
                  else RELEASE(Cell);
              end;
        end;
    end;
end.
```

Figure 2.7: The *GFC* procedure executed by processor p_i

```
procedure FIND-HEAD(Cell);
    Head ← ⊥;
    while Head = ⊥ and Cell.Next = ⊥ do begin
        for all cells TmpCell in memory do begin
            GRAB(TmpCell);
            if "success" and TmpCell.Next ≠ ⊥ and TmpCell.NotHead = false
              then Head ← TmpCell;
              else RELEASE(TmpCell);
        end;
    end;
end.
```

Figure 2.8: The *Find-Head* procedure.

allocated.

Proof: Assume that the lemma is false. Consider $2n^2$ cells that were allocated after p_i announced that it is trying to get a free cell. There exists at least one processor p_j that has allocated at least $2n$ cells among these $2n^2$. Observe, that a processor, during a single invocation of GFC procedure, can allocate at most n cells – one per each participating processor. Thus, since p_i has announce that it is looking for a free cell, p_j has executed GFC procedure from the beginning to the end at least once, which means that one of the $2n$ cells it has allocated has *ProcID* being equal to i, *i.e.* it belongs to p_i. This contradicts the assumption that there were no cells allocated for p_i. ∎

Implementation of APPEND The idea is to use the *Prev* field, which is an atomic Sticky-Byte object, to decide which processor succeeds with appending his cell to the list. The code of APPEND is given in Figure 2.9. First, look for the current head of the list using procedure FIND-HEAD, which code is given in Figure 2.8. To find the current head, scan all the cells looking for a cell with *Next* field defined, but with *NotHead* being false. This uses the fact that while appending a new cell C to the head H of the list, we first define $H.Prev$ to point to C, then we define $C.Next$ to point to H and only then we set $H.NoHead$ to true. Observe, that if during a single scan no cells were appended to the list, we will find the head. We will show below that from the moment processor p_i sets $AnnounceAppend(i)$, there are at most n cells appended to the list before its cell is appended. Therefore, after at most n scans of all the cells, we will either find the head of the list or will recognize that the *Cell* was already appended by

```
procedure APPEND(Cell);
    AnnounceAppend($i$) $\leftarrow$ 1;
    Head $\leftarrow$FIND-HEAD(Cell);
    Head $\leftarrow$ APPEND-INNER(Cell, Head, $i$);
    AnnounceAppend($i$) $\leftarrow$ 0;
    for all $j$ such that AnnounceAppend($i$) = 1 do begin
        Cell' $\leftarrow$ the cell processor $p_j$ is trying to append;
        XiffHead = $\perp$ Head $\leftarrow$ FIND-HEAD(Cell') Head $\leftarrow$ APPEND-INNER(Cell', Head, $j$);
    end;
    RELEASE(Head);
end.

procedure APPEND-INNER(Cell, Head, ID);
    if Cell.Next $\neq\perp$ then return Head;
    OldHead $\leftarrow$ Head.Next;
    while Cell.Next $=\perp$ do begin
        GRAB(OldHead);
        if "success" then jam OldHead.NotHead with "true";
        try jamming Head.Prev with pointer to Cell;
        RELEASE(OldHead);
        OldHead $\leftarrow$ Head;
        Head $\leftarrow$ Head.Prev;
        GRAB(Head);
        if "success" and (Head = Cell or Cell.Next $=\perp$) then Head.Next $\leftarrow$ OldHead;
    end;
    return Head;
end.
```

Figure 2.9: The *Append* procedure.

some other processor.

After finding the *Head* of the list, processor p_i checks whether the cell it tries to append is already in the list. If not, it jams the *Next* field of the head with the pointer to the second cell of the list and tries to jam the pointer to *Cell* into the *Prev* field of the head. If success, it writes the pointer to the head into the *Next* field of *Cell*, and indicates that *Head* is already not the head of the list by setting the *NoHead* field.

A failure to jam the *Prev* field with the pointer to the *Cell* means that some other processor succeeded in appending another cell C' to the list. Before writing anything into this cell, p_i tries to GRAB it in order to prevent initialization while writing. If the GRAB failed (or returned "empty"), it means that C' was already removed from the list. But C' is removed only if there are at least n cells ahead of it (see Figure 2.3). Hence, if GRAB fails, *Cell* was already added to the list by some other processor. By the same argument, if the GRAB succeeded, and the *Cell* is not yet in the list, then there are less than n cells in the list ahead of C', and therefore it could not have been initialized in between the moment it was added to the list and the GRAB. Hence, it is safe to write into it.

Before returning, p_i checks which processors have announced that they want to add their cell to the list, but have not succeeded in doing so yet. For each such processor p_j, processor p_i checks if the cell that p_j is currently trying to append is already in the list, and if not it appends it as if it was its own cell, by executing the above algorithm.

Lemma 2.6.2 At most n cells can be appended to the list from the moment a processor announces that it is trying to append a cell until this cell is actually appended.

Proof: Assume that the lemma is false. Consider n cells that are appended after p_i announces that it is trying to append a cell. There exists at least one processor that "owns" two cells among them. After appending the first one of these cells and before appending the second one, this processor has to help all other processors that are trying to append a cell. In particular, it has to help p_i, which leads to a contradiction. ▮

Space complexity The following theorem states the space complexity of our construction.

Theorem 2.6.3 **Any sequential object can be atomically implemented from $O(n^2 \log n)$ sticky bits and $O(n^2)$ cells, where each cell is large enough to hold a state of the object.**

Proof: We allocate the cells from a central pool, and therefore it is enough to count the number of cells which are in the list plus the cells which are being initialized. As we have already pointed out in the previous section, the number of cells in the list which are not being initialized is $O(n^2)$. Each processor might allocate at most a single cell during execution of GFC command per each other processor. In addition, each processor GRABS at most 3 cells at any moment, preventing their initialization. ▮

2.7 Conclusions and Open Problems

We have described a new approach to implement wait-free shared data objects in a shared-memory multiprocessor, based on the idea one processor helping another. We have introduced the Sticky Bit object, which can be viewed as a memory-oriented generalization of consensus, and showed how to use it to implement this approach. In particular, we showed how to construct a wait-free atomic implementation of any sequential object in terms of $O(n^2 \log n)$ Sticky Bits, where n is the number of processors.

Dolev, Dwork, and Stockmeyer [35] and Chor, Israeli, and Li [27] showed that there is no wait-free implementation of 1-bit Test-&-Set by safe registers. Furthermore, Herlihy [69] and Loui and Abu-Amara [92] showed that there is no wait-free protocol that achieves 3-processor consensus using atomic k-bit Test-&-Set for arbitrary k. Using the formalism presented in Section 2.2, it is easy to extend these proofs and show that no atomic Read-Modify-Write (RMW) operation on a single bit is powerful enough to implement a wait-free 3-processor consensus. On the other hand, observe that 3-processor consensus can be trivially achieved using a wait-free atomic RMW on 2 bits. This indicates that there exists a *RMW-hierarchy*, where safe-bits are on the bottom, 1-bit RMW is on the first level, 2-bit RMW is on the second level, etc. The universality of Sticky-Bit proves that the *RMW hierarchy collapses* at the 3rd level, because an atomic Sticky-Bit is trivially simulated by an atomic 2-bit RMW.

The results presented in this chapter suggest a number of natural direction for further research.

- Though the construction presented in Section 2.6 uses polynomial amount of memory, it is clearly not efficient. Is it possible to improve the memory and time complexities of this construction ?
- Can we prove a lower bound on the amount of memory needed to implement an arbitrary wait-free atomic object directly from safe bits ?
- Are there any other natural objects, besides Sticky Bits, that are universal, easily implementable in hardware, and convenient to use when programming shared-memory multiprocessors ?

Chapter 3

Local Management of a Global Resource in a Communication Network

3.1 Introduction

Consider a distributed protocol that has an acceptable message complexity under some reasonable assumptions (for example, a constant-time transmission delay assumption), but has a large or unbounded message complexity if these assumptions are not met. In this case it is desirable to add some mechanism that terminates the protocol if it sends too many messages. We discuss a natural generalization of this problem, *i.e.*, construction of a general *Resource Controller* that allows us to redirect or terminate a distributed protocol according to the amount of consumed resource.

Other examples when a Resource Controller is needeá include protocols with unacceptable worst case complexity but good average behavior in practice. For example, if a particular execution of a randomized algorithm uses too many resources it is usually better to abort the execution and to start the algorithm all over again. A Resource Controller may also be used to prevent untested algorithms in which errors may lead to explosive behavior, from crashing the network. Intuitively, the Resource Controller may be regarded as an "insurance policy", where the added message complexity of the protocol corresponds to the "premium", and the

[0] The section represents joint work with Y. Afek, B. Awerbuch, and M. Saks [3].

problem in [5]. The controller algorithms presented here build upon the techniques developed in these papers, with the addition of a feedback mechanism that enables us to restrict as well as monitor the resource consumption. Conversely, the controller enables us to construct simple and uniform solutions to the various forms of the counting problem.

Section 3.2 reviews the basic definitions and the model. We define the Resource Controller abstraction in Section 3.3. Section 3.4 presents an algorithm to implement a special case of the abstraction, the BASIC CONTROLLER, and Section 3.5 shows how to use this controller as a building block to implement a full-fledged Resource Controller. In Section 3.6 we describe the following two applications.

- *Dynamic Name Assignment.* Informally, the problem is to dynamically assign short unique names (or ID's) to the nodes participating in some protocol. By "short" we mean that the number of bits needed to represent a name is at most an additive factor more than the logarithm of the number of participating nodes. We assume that nodes join the protocol dynamically, and hence there is no way to preassign the names.

- *Distributed Bank.* The problem is to monitor a pool of resources, where nodes may withdraw resources from the pool and deposit new resources into the pool.

3.2 Model

We consider the standard point-to-point message passing asynchronous communication network model (see *e.g.* [46, 11]). The network topology is described by an undirected *communication graph*, where the set of nodes represents processors of the network and the set of edges represents bidirectional noninterfering communication channels operating between neighboring nodes. No common memory is shared by the node's processors, and there is no notion of a global clock. Messages sent over a link incur an arbitrary but finite delay.

The following complexity measures are used to evaluate the performance of distributed algorithms. The *Node Complexity* is the maximum number of nodes participating in the algorithm. The *Communication Complexity*, is the worst case total number of elementary messages sent during the algorithm execution, where an elementary message contains at most $O(\log n)$ bits, problem in [5]. The controller algorithms presented here build upon the techniques developed in these papers, with the addition of a feedback mechanism that enables us to restrict as well as monitor the resource consumption. Conversely, the controller enables us to construct simple and uniform solutions to the various forms of the counting problem.

Section 3.2 reviews the basic definitions and the model. We define the Resource Controller abstraction in Section 3.3. Section 3.4 presents an algorithm to implement a special case of the abstraction, the BASIC CONTROLLER, and Section 3.5 shows how to use this controller as a building block to implement a full-fledged Resource Controller. In Section 3.6 we describe the following two applications.

- *Dynamic Name Assignment.* Informally, the problem is to dynamically assign short unique names (or ID's) to the nodes participating in some protocol. By "short" we mean that the number of bits needed to represent a name is at most an additive factor more than the logarithm of the number of participating nodes. We assume that nodes join the protocol dynamically, and hence there is no way to preassign the names.

- *Distributed Bank.* The problem is to monitor a pool of resources, where nodes may withdraw resources from the pool and deposit new resources into the pool.

3.2 Model

We consider the standard point-to-point message passing asynchronous communication network model (see *e.g.* [46, 11]). The network topology is described by an undirected *communication graph*, where the set of nodes represents processors of the network and the set of edges represents bidirectional noninterfering communication channels operating between neighboring nodes. No common memory is shared by the node's processors, and there is no notion of a global clock. Messages sent over a link incur an arbitrary but finite delay.

The following complexity measures are used to evaluate the performance of distributed algorithms. The *Node Complexity* is the maximum number of nodes participating in the algorithm. The *Communication Complexity*, is the worst case total number of elementary messages sent during the algorithm execution, where an elementary message contains at most $O(\log n)$ bits, where n is the number of participating nodes (as opposed to the total size of the network, which is denoted by V). The *Bit Complexity* is the worst case total number of bits sent during the algorithm execution.

3.3 Resource Controller – Definition

A Resource Controller is a distributed algorithm executing concurrently with the algorithm whose resource consumption is being controlled, which we call the *controlled* algorithm. Intuitively, each unit of resource used by the controlled algorithm must be authorized in advance by the Resource Controller. Formally, we require that as part of the controller, each node has a procedure called RESOURCE-REQUEST, whose code depends on the specific controller and the identity of the node executing it. When a node executing the controlled algorithm needs a unit of resource, it must call this procedure and wait until a *permit* is returned. A request that never receives a permit is said to be *blocked.* Otherwise, we say that the request was *satisfied.*

A Resource Controller with parameters M and W satisfies the following requirements:

1. At most M requests are satisfied.
2. The controller terminates by producing a signal at a distinguished node if and only if there is a blocked request.
3. If the Resource Controller terminates then at least $M - W$ requests were satisfied (*i.e.*, if M resources are available, at most W are "wasted" - not granted at termination).

To simplify the description of the algorithms, we make the following assumptions:

- The controlled algorithm is a single initiator distributed algorithm with new nodes dynamically joining the set of participating nodes.
- The graph induced by the participating nodes is a tree rooted at the initiator. Every node knows the length of the path from it to the root through the edges of the tree.

This model abstracts the situation in which a single-initiator protocol is executed in a large network. These assumptions can be easily satisfied by an auxiliary process which *dynami-*

cally constructs a spanning tree of the participating nodes as it was done by Dijkstra and Scholten [33].

3.4 The Basic Controller.

This section describes a Resource Controller called the Basic Controller, which operates under the assumption that an upper bound U on the number of participating nodes is known. In the following section we show how to relax this assumption, using the Basic Controller as the main building block.

Intuitively, the Basic Controller propagates the resource requests up to the root and distributes the permits from the root. A single message can represent a number of resource requests or a number of permits. The main idea of the algorithm is to aggregate a large number of requests/permits together, represent them by a single message, and allow this message to travel a distance which is proportional to this number.

To achieve the aggregation we use a data structure of Afek and Saks [5], which defines a hierarchy on the nodes of the spanning tree. Each node is assigned to a *level* in this hierarchy. (The level of a node is different from its depth in the tree.) In addition, each node v is assigned a supervisor, $Supervisor(v)$, which is either the root of the spanning tree or the nearest ancestor of v at level $Level(v) + 1$.

An example of such a hierarchy for the case of a chain is presented in Figure 3.1. Numbers inside the circles represent the depth of nodes, *i.e.*, their distance from the root; numbers inside rectangular boxes represent levels.

More precisely, let the depth $D(v)$ of v be the length of the path from v to the root of the spanning tree. Define $Level(v) = \max\{i \mid 2^i \text{ divides } D(v)\}$. Define $Supervisor(v)$ to be either the root of the spanning tree, or the closest ancestor of v whose level is one higher than the level of v, whichever is closer. For example, if depth of v is 52 (binary 110100), $Level(v) = 2$ and $Supervisor(v)$ is the ancestor of v at depth 40 (binary 101000). The above hierarchy has the following properties, which are central to its use in the controller.

Lemma 3.4.1

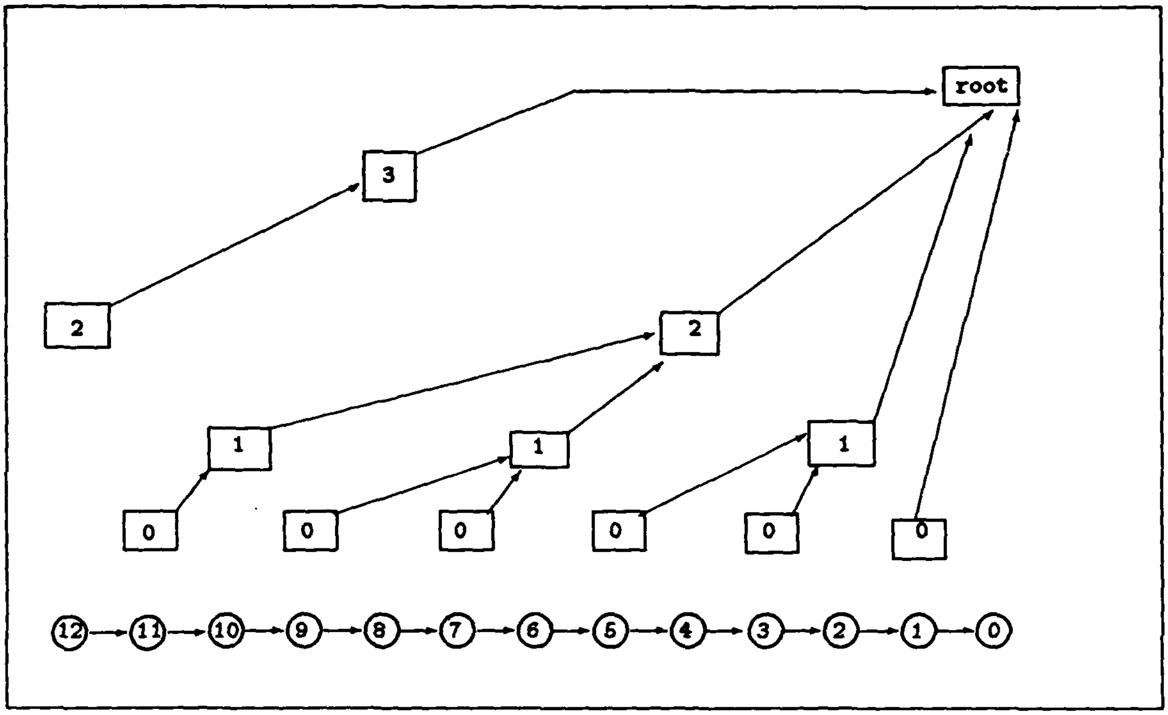

Figure 3.1: An example of a hierarchy on a chain. Circled nodes represent the original graph; rectangular nodes form the hierarchy.

1. The number of nonempty levels is at most $\log U$,

2. If v is at level l then the path from v to *Supervisor*(v) has either 2^l or $3 \cdot 2^l$ links,

3. The number of nodes at level l that are supervisors is at most $U/2^{l-1}$.

Proof: The first two parts follow from the definition. To prove the third part, for each supervisor v at level l, consider a path from u to v where v is the supervisor of u. Clearly, these paths are disjoint and each contains at least 2^{l-1} vertices. ▮

As with every controller, the Basic Controller has a Resource-Request procedure at every node. In addition, every node except the root has a process, called Delivery-Process, and the root has a Root-Process. Intuitively, the Basic Controller works as follows. Each node has two bins where permits can be stored. The first bin is managed by the Delivery-Process, and the second bin is managed by the Resource-Request procedure. The bin capacity of the Delivery-Process depends on the level of the node. The bin capacity of the Delivery-

PROCESS located at node v is either 1 or half the capacity of the DELIVERY-PROCESS bin at node *Supervisor*(v), whichever is larger. The capacity of the RESOURCE-REQUEST bin is either 1 or half the capacity of the DELIVERY-PROCESS bin of a node at level 0, whichever is larger. To ensure that at most W resources are "wasted", we choose the capacities in such a way that the sum of the capacities of DELIVERY-PROCESS bins at supervisor nodes and all RESOURCE-REQUEST bins is at most W.

Initially all bins are empty except the bin of the root, which contains M permits. When the controlled protocol at some node calls the procedure RESOURCE-REQUEST, the procedure issues a permit if it has one in its bin. Otherwise, before issuing a permit, it sends a message to the local DELIVERY-PROCESS, asking it to relay the request to replenish the bin to the nearest node at level 0 up the spanning tree.

In general, if the bin of the DELIVERY-PROCESS at node v is empty when it receives a request, it first sends a request addressed to the DELIVERY-PROCESS at node *Supervisor*(v) for enough permits to replenish its bin, and waits for these permits before replying to the outstanding request. Each node on the path from v to *Supervisor*(v) serves as a relay for the communication between v and *Supervisor*(v). Any message of the algorithm is either a request propagating up the tree or a permit going down. After satisfying a request, a DELIVERY-PROCESS bin is either half-full or empty. A request arriving at the root when the DELIVERY-PROCESS bin at the root is empty generates the termination signal for the BASIC CONTROLLER algorithm.

More precisely, let

$$\Lambda = \left\lfloor \log_2 \left(\frac{W}{U} \cdot \frac{1}{2 \log U} \right) \right\rfloor .$$

Then we define the capacity of the DELIVERY-PROCESS bin at level l to be

$$\text{Cap}(l) = \max\{2^{l+\Lambda}, 1\}.$$

The capacity of each RESOURCE-REQUEST bin is defined as if it is a DELIVERY-PROCESS bin of a node at level -1.

Lemma 3.4.2 **The message complexity of the** BASIC CONTROLLER **is** $O(\frac{M}{W} U \log^2 U)$.

Proof: The algorithm sends only two types of messages: requests and permits. There may be more request messages than permit messages, but clearly the number of request messages sent

GET-NEXT-MESSAGE: Dequeues the message at the top of the queue and returns two variables, the first is the message (MSG), and the second is the number of the link over which the message came (LINK#). If the queue is empty, GET-NEXT-MESSAGE waits until a message arrives. Formally, the syntax is: *MSG,LINK#* ← GET-NEXT-MESSAGE.

WAIT-FOR-MESSAGE-FROM-LINK*(Link#)*: Removes from the queue a message which came via link number *LINK#*. If there is no such message in the queue it waits until such a message arrives. Note that any other message that arrives while waiting is queued in the central queue.

SEND-MESSAGE*(MSG,Link#)*: Sends the message *MSG* through link *Link#*.

Figure 3.2: Description of the communication primitives.

```
/* Initialization */
PERMITS-IN-BIN ← 0;                                        /*Initially the bin is empty.*/
Max-Permits-In-Bin ← max{1, $2^{\Lambda-1}$};                /*RESOURCE-REQUEST bin capacity.*/

Procedure RESOURCE-REQUEST
    if PERMITS-IN-BIN = 0
      then begin
            SEND-MESSAGE(−1,Internal-Link);                 /*Ask for replenishment.*/
            WAIT-FOR-MESSAGE-FROM-LINK(Internal-Link);      /*Wait for the permits.*/
            PERMITS-IN-BIN ← Max-Permits-In-Bin;
            end;
    PERMITS-IN-BIN ← PERMITS-IN-BIN - 1;
    return;                                                 /*Issue a permit.*/
end.
```

Figure 3.3: The code of the RESOURCE-REQUEST procedure

```
Procedure DELIVERY-PROCESS
/*Initialization for node whose distance in the tree from the root is $D$. */

My-Level ← max$\{i \mid 2^i$ divides $D\}$;            /* The level of the node in the hierarchy */
Bin-Capacity ← max$\{1, 2^{\text{My-Level}+\Lambda}\}$;  /*Bin capacity of the Delivery-Process bin at node $v$.*/
BIN-EMPTY ← true;                                                  /*Initially, the bin is empty.*/

Do forever
    LEVEL,LINK ← GET-NEXT-MESSAGE           /*Wait for the next request, which originated from a*/
                                            /*node at level LEVEL and arrived on link# LINK.*/
    if My-Level = LEVEL +1
      then begin                                     /*You are the supervisor of the requesting node,*/
                                                           /*try to satisfy the request from your bin.*/
           if BIN-EMPTY                               /*Test whether you can satisfy the request.*/
             then begin                                            /* You need a replenishment. */
               SEND-MESSAGE(My-Level, Parent-Link);                  /*Request bin replenishment.*/
               WAIT-FOR-MESSAGE-FROM-LINK(Parent-Link) ;          /*Wait for bin to be replenished.*/
               if Bin-Capacity > 1 then BIN-EMPTY ← false ;            /* Store permits that will */
                                                                      /* not be sent immediately. */
               end;
             else BIN-EMPTY ← true;                                  /* Remove permits from bin.*/
           SEND-MESSAGE("permit",LINK);
           end;
      else begin
          SEND-MESSAGE(LEVEL,Parent-Link);                       /*Relay the request to the parent.*/
          WAIT-FOR-MESSAGE-FROM-LINK(Parent-Link);                          /*Wait for the permit,*/
          SEND-MESSAGE("permit",LINK);                                        /*and relay it down.*/
          end;
    end;
end.
```

Figure 3.4: The code of the DELIVERY-PROCESS procedure.

```
Procedure ROOT-PROCESS
    BIN ← M ;                                          /* All the permits we have.*/
    do forever
    LEVEL,LINK ← GET-NEXT-MESSAGE ;
    if BIN > 0
      then begin
            RQST-CAPACITY ← $\max\{1, 2^{\text{LEVEL}+\Lambda}\}$;      /*The number of permits requested.*/
            BIN ← BIN - RQST-CAPACITY ;
            SEND-MESSAGE("permit",LINK);
            end
      else STOP, and generate Signal;                  /* The Root-Process bin is empty.*/
    end;
end.
```

Figure 3.5: The code of the ROOT-PROCESS procedure.

from u to *Supervisor*(u) is at most one more than the number of permit messages sent from *Supervisor*(u) back to u. Therefore it suffices to bound the number of permit messages.

The total number of permits is at most M. By definition, the number of permits in one message received by a supervisor at level l is exactly the size of the bin at level l, namely *Cap*(l). Hence, the total number of permit messages received by the supervisors at level l is at most $\frac{M}{Cap(l)}$. By Lemma 3.4.1, each one of these messages travels a distance of at most $3 \cdot 2^l$. Hence, summing over all $(\log U + 1)$ levels (for this computation we can treat the RESOURCE-REQUEST procedures as if they are DELIVERY-PROCESSES at level -1), we get $O(\frac{M}{W} U \log^2 U)$. ▮

Since all bins are initially empty, and all the permits originate from the root, the total number of permits given out by the BASIC CONTROLLER is at most M. The correctness of the algorithm follows from the following lemma.

Lemma 3.4.3 At least $M - W$ resource requests are satisfied if any request is blocked.

Proof: By the previous Lemma there is a finite number of messages sent by the BASIC CONTROLLER. Consider the moment when the last one of these messages is received. At this time every permit has either been issued or is stored in bins of non-unit capacity. If a request is blocked, the bin of the root is empty. Hence, the number of issued permits is at least M minus the total number of permits stored in the rest of the bins. A bin may be non-empty only if it belongs to a node which is a supervisor. Hence, by Lemma 3.4.1, the total capacity of bins with capacity of at least 2 on level l is at most

$$\frac{U}{2^{l-1}} \cdot 2^{l+\Lambda} \leq \frac{W}{\log U}$$

The number of levels with bin capacities of at least 2 is at most $\log U$ and therefore the sum of bin capacities is at most W. ▮

Recall that the only messages received by a node are requests from its children or permits from its parent. To minimize bit complexity and space requirements we require strict alternation between permit and request messages on every link. Thus every time a node sends a request to its parent it waits for a permit to come back before sending the next request. The node queues any request message it receives while waiting for a permit. A request message propagating up the tree from v to *Supervisor*(v) consists only of a single field, containing the level of v. When a request form v arrives at *Supervisor*(v) all the nodes on the path between them are waiting for a permit message to come back. Message size is thus at most $O(\log\log U)$ bits. Note, that every node needs to store at most one elementary message per link. Therefore,

Corollary 3.4.4 **The bit message complexity of the** BASIC CONTROLLER **is** $O(\frac{M}{W} U \log^2 U \log\log U)$ **and memory requirements are** $O(\log\log U)$ **bits per link.**

The code of the BASIC CONTROLLER is shown in Figures 3.3, 3.4, and 3.5. The bins are represented explicitly only in the root of the spanning tree and in the procedure RESOURCE-REQUEST. The rest of the bins are implicit, in the sense that if a node has a bin of capacity ≥ 2, we know that it can satisfy exactly 2 requests from the nodes it supervises. A node with bin capacity 1 can satisfy only one request; hence it merely serves as a relay of requests and permits between the nodes it supervises and its supervisor.

Each node maintains a central queue in which arriving messages are stored in the order received. Basic communication primitives are described in Figure 3.2. *Parent-Link* is the link connecting the node to its father in the spanning tree. The *Request-Resource* procedure communicates with the DELIVERY-PROCESS at the same node though a logical link, called the *Internal-Link*.

3.5 Main Controller

The previous section presented the Basic Controller algorithm which assumed an a-priori knowledge of an upper bound on the number of participating nodes. The complexity of the Basic Controller is a function of this upper bound regardless of the actual number of participating nodes. In this section we show how to relax this assumption and describe the Main Controller whose complexity depends only on the actual number of participating nodes.

The basic idea is to run two Basic Controllers concurrently. Controller-N monitors and controls the number of participating nodes; Controller-R monitors and controls the resource consumption. We require that a new node that wishes to join the controlled protocol does not participate until it gets a permit to do so from Controller-N.

The algorithm proceeds in iterations. The number of nodes that are allowed to join during iteration i is at most twice the number of nodes in the beginning of the iteration. We use Controller-N to terminate the iteration when the number of nodes has at least doubled compared to the number of nodes in the beginning of the iteration.

More precisely, assume that at the beginning of iteration i the number of participating nodes is n_i. We set (M, W) of Controller-N to $(3n_i, n_i)$. By Lemma 3.4.3, when Controller-N terminates, the number of participating nodes is $3n_i \geq n_{i+1} \geq 2n_i$. As long as it does not terminate, both Controller-N and Controller-R use $3n_i$ as the upper bound on the number of participating nodes.

When Controller-N terminates at the root, it executes a "broadcast and echo" on all the participating nodes, collecting the contents of their bins (both the Controller-N and Controller-R bins). When the echo terminates, the number of participating nodes, n_{i+1}, and the number of unused resources M_{i+1} are known at the root. If $M_{i+1} \leq W$ the algorithm terminates, otherwise both controllers are restarted with bin sizes corresponding to the new upper bound $3n_{i+1}$, and with Controller-R using (M_{i+1}, W) as its parameters.

The correctness of the Main Controller follows from the correctness of the Basic Controller and from the fact that at any point we maintain a constant factor approximation to the number of participating nodes. Let n be the final number of participating nodes.

Theorem 3.5.1 The message complexity of the MAIN CONTROLLER is $O(\frac{M}{W} n \log^2 n)$ and the bit message complexity is $O(\frac{M}{W} n \log^2 n \log\log n)$.

Proof: The message complexity of the MAIN CONTROLLER is the sum of the message complexities of CONTROLLER-N and CONTROLLER-R. In iteration i, these complexities are $O(n_i \log^2 n_i)$ and $O(\frac{M_i}{W} n_i \log^2 n_i)$, respectively. By definition, $M \geq M_i \geq W$. Moreover, for each i, $3n_{i-1} \geq n_i \geq 2n_{i-1}$. The bound follows. Note that the communication complexity of collecting the contents of all the bins is at most $O(n \log n)$ messages or $O(n \log^2 n)$ bits. ▮

Before proceeding to the applications, let us mention three extensions to the controller:

1. The $W = 0$ case. Some applications require a zero "waste". Recall that the size of the bins of the controller is inversely proportional to W which prevents us from applying the MAIN CONTROLLER directly. Also observe that applying the MAIN CONTROLLER in cases where W is small compared to M leads to large complexity. In general, in order to deal with cases where M/W is large, we iterate the MAIN CONTROLLER $\log M$ times. In each iteration the "waste" is at least halved. That is, we set $M_0 = M$. In the ith iteration we execute the MAIN CONTROLLER with parameters $(M_i, M_i/2)$. When it terminates the root performs a "broadcast and echo" to count the number of unused resources, which is $M_{i+1} \leq M_i/2$. By Theorem 3.5.1, the message complexity of the above algorithm is $O(n \log^2 n \log M)$.

2. Unknown M case. In some applications the maximum number of resources available is not known in advance. In such cases we invoke the MAIN CONTROLLER iteratively, setting $M = 2^i$ and $W = M/2$ at the beginning of i-th iteration. At the end of each iteration we allow an external mechanism to decide whether the ultimate limit was reached and to terminate the algorithm. Observe that the sequence of iterations of the MAIN CONTROLLER interacts with a single execution of the controlled algorithm. By Theorem 3.5.1, the message complexity of this algorithm is $O(I \cdot n \log^2 n)$, where I is the number of iterations and n is the final number of participating nodes.

3. The Resource Controllers described thus far assume that the set of participating nodes dynamically grows. That is, once a node has joined the set it may not leave it. If we

apply the Main Controller directly, the complexity will be in terms of the total number of participating nodes n, which might be much larger than the number of nodes actually active at any point of time. It is easy to generalize the algorithms to the case in which the set of participating nodes dynamically grows and shrinks, as long as the subnetwork induced by this set is connected and we know an upper bound n_{max} on the size of the set of concurrently active nodes. The idea is to control the number of active nodes to within a constant of n_{max}. This is done by a separate Resource Controller which approximates the number of inactive and the number of active nodes in the tree. Whenever the number of inactive nodes in the spanning tree exceeds n_{max}, the tree is reconstructed using only the set of currently active nodes.

3.6 Applications

The motivating application of the Main Controller is to control the worst case message and node complexities of a distributed algorithm, as described in the introduction. The techniques described in the previous sections can be applied to solve several related problems, two of which are presented in this section.

3.6.1 Dynamic Name Assignment

Many protocols assume that each node in a network has a unique name (ID) and use these names to break symmetry (for example, see Gallager, Humblet, and Spira [46], Afek, Landau, Schieber, and Yung [4], as well as Chapter 1 of this thesis). The bit message complexity of these protocols is expressed in terms of the number of bits needed to represent a name, which is usually assumed to be equal to the logarithm of the number of nodes. This assumption is correct only if at least a constant fraction of the total number of the nodes are participating in the protocol, which is not always true.

We define the *Dynamic Name Assignment* problem as follows. As in the case of the Main Controller, we consider a single-initiator protocol (the extension to multiple initiators is not difficult) that executes in a large network and dynamically activates new nodes, which start participating in the protocol. The goal is to assign unique integer names to all the participating nodes, such that the largest name will be at most a constant factor larger than n, the number of participating nodes.

To solve this problem we use CONTROLLER-N with the following change: the "permit messages" carry the *range of allocated names* instead of carrying permits. Note, that as opposed to the case of CONTROLLER-N, every message in the Dynamic Name Assignment algorithm carries a range of names, and *hence it is* $O(\log n)$ bits long. Moreover, since only contiguous ranges can be represented concisely, we can not reuse names that are in the bins when CONTROLLER-N is initialized at the beginning of each iteration, which increases the number of unused names. Observe, that this increase is by at most a constant factor.

Lemma 3.6.1 **The message complexity of the Dynamic Name Assignment algorithm is $O(n \log^2 n)$ with messages of $O(\log n)$ bits, where n is the final number of participating nodes.**

Proof: Similar to Lemma 3.5.1. ▮

3.6.2 Distributed Bank

The MAIN CONTROLLER described in the previous sections deals with resources that can be only consumed. Here we extend the MAIN CONTROLLER to the case where resources are both consumed and generated.

We make assumptions similar to the ones in Section 3.3. In addition we assume that the nodes participating in the controlled protocol may request to *withdraw* or to *deposit* a unit of resource. A Distributed Bank Controller with parameter W is a distributed algorithm which interacts with the controlled algorithm via the DEPOSIT-RESOURCE and WITHDRAW-RESOURCE procedures at the nodes, and satisfies the following requirements:

1. At any time, the total number of withdrawal requests granted is at most the total number of deposit requests.

2. If a withdrawal request remains unfulfilled forever, then the balance (the number of deposits less the number of granted withdrawals) eventually falls below W and stays there.

A Distributed Bank Controller can be built by combining two MAIN CONTROLLERS, where one controls the withdraw requests and the other controls the deposit requests. The two controllers exchange resources at the root. That is, all the excess resources that are collected by the deposit controller, are passed to the bin of the withdrawal controller at the root.

Another, somewhat more practical solution, is to regard deposit requests as "positive" and withdrawal requests as "negative". In other words, the response to a withdrawal request is the same as in the MAIN CONTROLLER; if a deposit request arrives to an empty or half-full bin, then the request is added to the bin. If the bin is full, the excess is sent up to the supervisor as a deposit request at the corresponding level. This solution, though more practical, raises some fairness issues which are beyond the scope of this thesis.

Let $\mathcal{M}$ be the total number of withdraw and deposit requests made.

Lemma 3.6.2 The message complexity of the Distributed Bank Controller is $O(\frac{\mathcal{M}}{W} n \log^2 n)$ and the bit complexity is $O(\frac{\mathcal{M}}{W} n \log^2 n \log\log n)$.

Proof: Similar to the proof of Lemma 3.4.2 and Theorem 3.5.1. ▮

3.7 Conclusions

The search for the right set of paradigms for designing efficient distributed algorithms is a fundamental task of the theory of distributed computation. Afek, Awerbuch, and Gafni [2] and Awerbuch [12] have already used the Resource Controller to design several important algorithms, and we believe that it is potentially applicable to many diverse problems in distributed computation.

Chapter 4

Minimum-Cost Spanning Tree as a Path-Finding Problem

4.1 Introduction

Linear arrays and mesh-connected systolic arrays are an important class of parallel computers. The simplicity of their interconnection network makes them attractive from the hardware implementation point of view, but, on the other hand, presents the algorithm designer with a challenge to map the data in a way which minimizes nonlocal communication.

Many problems are solved by algorithms that can be naturally mapped into a mesh-connected computer. In particular, this is true for the path-finding problem in a closed semiring [7, sections 5.6–5.9]. For a graph of n vertices, the path-finding problem can be solved sequentially in $O(n^3)$ steps by a dynamic programming algorithm [80, 99] of which the algorithms of Floyd [40] and Warshall [129] are special cases. This dynamic programming algorithm has a well known $O(n)$-time implementation on an $n \times n$ mesh-connected computer [10, 28, 32, 67, 116].

Graph problems that can be cast as path-finding problems include transitive closure and all-pairs shortest paths. These instances differ only in the definitions of the operators of the closed semiring. Many other simple $O(n)$-time algorithms for mesh-connected computers are based on finding shortest paths, including the problem of finding bridges and articulation points, and

[0] This chapter represents joint work with Bruce Maggs [97].

the problem of finding a breadth first spanning tree [10, 32].

In this chapter we show that minimum-cost spanning tree is a special case of the closed semiring path-finding problem. Previously known minimum-cost spanning tree algorithms for the mesh [10, 90] are based on the recursive algorithm of Boruvka (also attributed to Sollin) [124, pp. 71–83], which is complicated to implement. For example, the algorithm of Atallah and Kosaraju [10] achieves $O(n)$-time by reducing the fraction of the mesh in use by a constant factor at each recursive call. Our dynamic programming algorithm has the same asymptotic running time but is much simpler.

The rest of this chapter consists of two short sections. In Section 2 we show how to cast minimum-cost spanning tree as a path-finding problem. In Section 3, we briefly describe an $O(n)$-time mesh algorithm to solve the problem.

4.2 Minimum-cost spanning tree

In this section we define the minimum-cost spanning tree problem and a related path-finding problem. We give a recurrence for solving the path-finding problem via dynamic programming. We then prove that the solution to the path-finding problem contains the solution to the minimum-cost spanning tree problem.

Given an n-node connected[1] undirected graph $G = (V, E)$, where V is the set $\{1, \dots n\}$, and where each edge $\{i, j\}$ in E has cost $C^0_{ij} = C^0_{ji}$, the minimum-cost spanning tree problem is to find a subgraph that connects the vertices in V such that the sum of the costs of the edges in the subgraph is minimum. We assume that the edge costs are unique. (If not, lexicographical information can be added to make them unique.) For convenience, we also assume that if $\{i, j\}$ is not in E then it has cost $C^0_{ij} = C^0_{ji} = \infty$.

The path-finding problem is to compute the cost C^k_{ij} for each $1 \leq i, j, k \leq n$ of the shortest (lowest-cost) path from i to j that passes through vertices only in the set $\{1, \dots, k\}$, where *the cost of a path is defined to be the highest cost of any edge on the path.* For any i and j, the shortest path from i to j with no intermediate vertex higher than k either passes through k or

[1] For simplicity, we assume that the graph is connected. The same technique will find a minimum-cost spanning forest of a disconnected graph.

does not. In the first case, the cost of the shortest path from i to j is either the cost of the shortest path from i to k or the cost of the shortest path from k to j, whichever is higher. In the second case, we have $C_{ij}^k = C_{ij}^{k-1}$. Thus, C_{ij}^k can be computed by the recurrence

$$C_{ij}^k = \min\{C_{ij}^{k-1}, \max\{C_{ik}^{k-1}, C_{kj}^{k-1}\}\}.$$

The following theorem shows that the unique minimum-cost spanning tree can be recovered from the costs of the shortest paths.

Theorem 4.2.1 An edge $\{i,j\}$ belongs to the unique minimum-cost spanning tree if and only if $C_{ij}^0 = C_{ij}^n$.

Proof: The proof has two parts. We first show that if $\{i,j\}$ is a tree edge then $C_{ij}^0 = C_{ij}^n$. We then show that if $C_{ij}^0 = C_{ij}^n$ then the edge $\{i,j\}$ is in the tree. First, assume that $\{i,j\}$ is a tree edge, but that $C_{ij}^0 \neq C_{ij}^n$. Consider the cut of the graph that $\{i,j\}$ crosses, but no other tree edge crosses. Since $C_{ij}^0 \neq C_{ij}^n$, there must be some path from i to j whose highest-cost edge has cost $C_{ij}^n < C_{ij}^0$. Hence, every edge on this path has cost less than C_{ij}^0. This path must cross the cut at least once. Replacing the edge $\{i,j\}$ by any edge on the path that crosses the cut reduces the cost of the tree, a contradiction. Conversely, assume that $C_{ij}^0 = C_{ij}^n$, but that $\{i,j\}$ is not a tree edge. Adding the edge $\{i,j\}$ to the tree forms a cycle whose highest-cost edge costs more than than C_{ij}^0. Replacing this edge by $\{i,j\}$ yields a tree with smaller cost, a contradiction. ▮

4.3 Implementation on a mesh-connected computer

In this section we give a short description of an $O(n)$-time algorithm for solving the minimum-cost spanning tree problem on an $n \times n$ mesh-connected computer. We assume that the diagonal element in each mesh row can broadcast a value to the other elements of the row in a single step. This type of broadcast can be simulated by a mesh without this capability by slowing the algorithm down by a constant factor [86, 88, 89]. Figure 4.1 presents an example of a retimed 4×4 mesh. Numbers near edges represent delays.

The algorithm proceeds as follows. We assume that the input graph is given in the form of a matrix of edge costs C^0 which enters row-by-row through the top of the mesh. Matrix row i is modified as it passes over rows 1 through $i-1$ and is stored when it reaches mesh row i. When matrix row i passes over mesh row k, the value C_{ik}^{k-1} is broadcast right and left from the diagonal cell (k,k). Each cell $(k,j), 1 \leq j \leq n$ knows the value of C_{kj}^{k-1} and computes

$$C_{ij}^{k} = \min\{C_{ij}^{k-1}, \max\{C_{ik}^{k-1}, C_{kj}^{k-1}\}\}.$$

which is passed down to the next mesh row. After reaching mesh row i, matrix row i stays there until each matrix row l, $i < l \leq n$, above it has passed over it and then continues to propagate down, passing over the rest of the matrix rows. The output matrix C^n exits row-by-row from the bottom of the mesh. By Theorem 1, the adjacency matrix of the minimum-cost spanning tree can be constructed by comparing the input and output matrices.

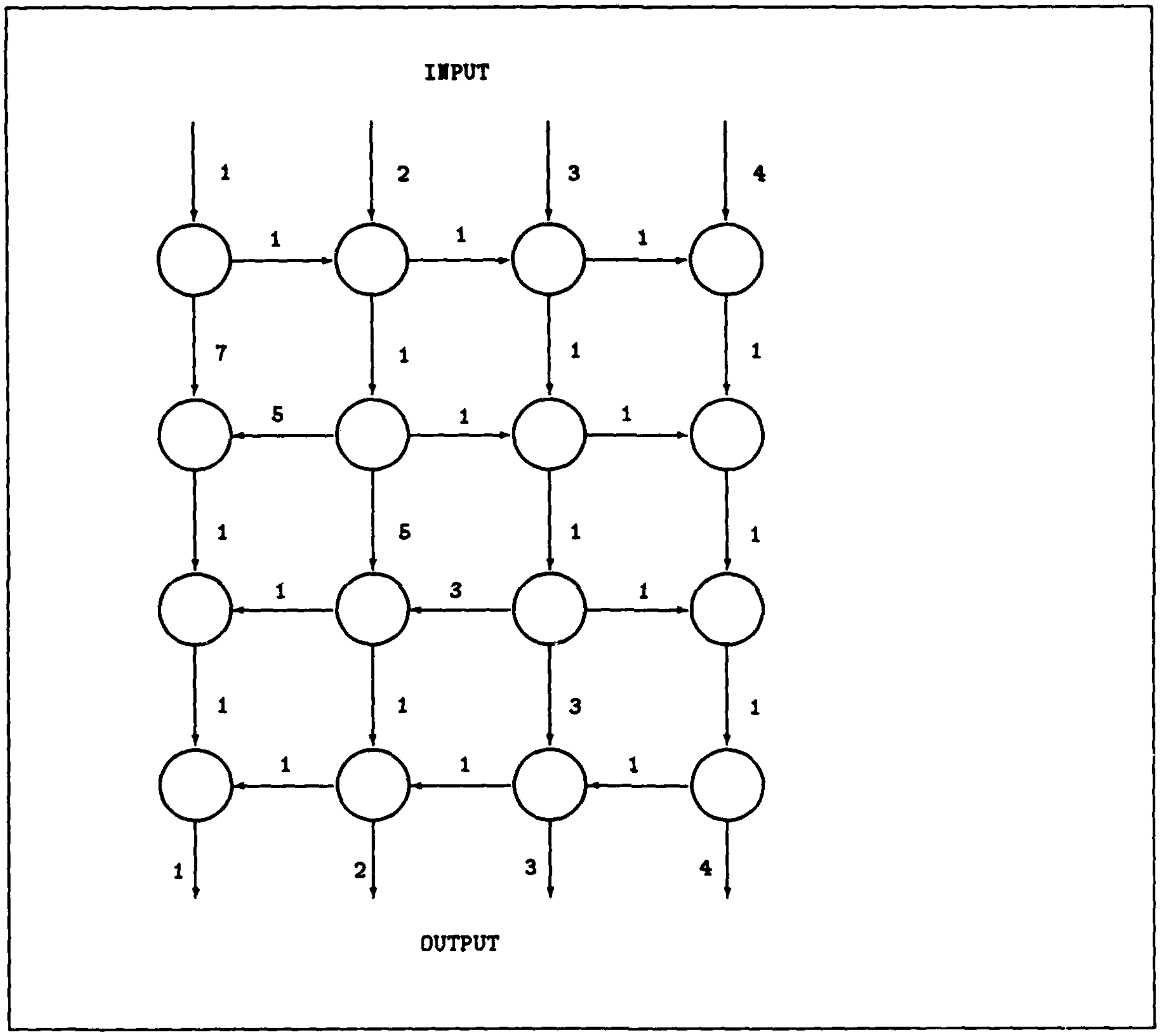

Figure 4.1: An example of a retimed mesh-connected computer for minimum-weight spanning tree computation. Numbers near the edges represent delays.

Chapter 5

Sublinear-Time Parallel Algorithms for Matching and Related Problems

5.1 Introduction

Bipartite matching and related problems have been studied extensively in the contexts of both sequential (*e.g.* [38, 71, 125]) and parallel (*e.g.* [6, 77, 102]) computation. Though recent research produced RNC algorithms for these problems, *i.e.*, randomized parallel algorithms that run in expected polylogarithmic time on a polynomial number of processors, no sublinear-time deterministic parallel algorithms were known. This chapter describes several techniques that allow us to construct such algorithms for bipartite matching, flows in zero-one capacity networks, depth-first search, and the problem of finding a maximal set of node-disjoint paths.

Our algorithms for bipartite matching and for zero-one flows generalize to weighted versions of these problems. These generalizations involve the technique of scaling, so the resulting algorithms run in sublinear time if the weights are polynomially bounded. Although Karp, Upfal, and Wigderson have given an RNC algorithm for the assignment problem with unary weights, until now no sublinear-time deterministic algorithms have been discovered.

The results presented in this chapter are based on a more complete understanding of the combinatorial structure of the above problems, which leads to new algorithmic techniques. In particular, we show how to use maximal matching to extend, in parallel, a current set of node-

[0] This chapter represents joint work with A. Goldberg and P. Vaidya [57, 56].

disjoint paths. We also show how to take advantage of the parallelism that arises when a large number of nodes are "active" during an execution of Goldberg's parallel push/relabel network flow algorithm [58].

Our model of parallel computation is a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) [42]. Given a directed graph with n nodes and m arcs, $BFS(n,m)$ denotes the maximum of $n+m$ and the number of processors required to find a breadth-first search tree in $O(\log^2 n)$ time; $SSP(n,m)$ denotes the maximum of $n+m$ and the number of processors required to find a single-source shortest-path tree (with nonnegative weights) in $O(\log^2 n)$ time. It is known that $SSP(n,m) \leq n^3$, and that $BFS(n,m)$ is at most the number of processors required to multiply two $n \times n$ matrices in $O(\log n)$ time, which is $O(n^{2.5})$ [108].

In this chapter we address the following problems. (Complexity of the algorithms is given in terms of the number of node n, number of edges m, and the largest absolute value of the weights (or costs) C.)

Maximal node-disjoint paths We are given a graph $G = (V, E)$ with a set of sources $S \subset V$ and a set of sinks $T \subset V$, such that $S \bigcap T = \emptyset$. A set Π of node-disjoint paths is said to be *maximal* if each path in Π starts at a distinct source and terminates at a distinct sink, and there is no path from a source to a sink in the graph induced by the nodes in $V - \Pi$. The maximal node-disjoint paths problem is to find a maximal set of node-disjoint paths from S to T. We give an algorithm for the maximal node-disjoint paths problem which runs in $O(\sqrt{n}\log^3 n)$ time, both on directed and on undirected graphs. On undirected graphs our algorithm uses $O(n+m)$ processors, and on directed graphs it uses $BFS(n,m)$ processors. The algorithm also solves a slight generalization of the maximal node-disjoint paths problem, which is useful for constructing a depth-first search tree in an undirected graph.

Depth-first search in undirected graphs Given an undirected graph $G = (V, E)$ and a distinguished node, construct a depth-first search tree of the graph rooted at this node. A tree T is a *depth-first search tree* if and only if for all nontree edges (u, v), nodes u and v lie on the same path starting at the root of the tree. The previous parallel algorithms of Anderson [9] and Aggarwal-Anderson [6] for the problem use randomization. Using our our techniques we convert the Aggarwal-Anderson randomized depth-first search algorithm into a deterministic algorithm that runs is $O(\sqrt{n}\log^5 n)$ time using $O(n+m)$ processors.

Bipartite matching Given an undirected bipartite graph $G = (S, T, E)$, where $S \bigcup T$ is the set of nodes ($S \bigcap T = \emptyset$) and $E \subset (S \times T)$ is the set of edges, find a maximum cardinality matching in G. We present an algorithm for the bipartite matching problem which runs in $O(n^{2/3}\log^3 n)$ time using $BFS(n, m)$ processors. Although Karp, Upfal, and Wigderson have shown that the problem belongs to RNC [77], the previous fastest deterministic algorithm, due to Schieber and Moran [115], runs in $\Theta(n)$ time in the worst case.

Assignment problem (Also known as the weighted bipartite matching problem.) Given a weighted undirected bipartite graph $G = (S, T, E)$, find a minimum weight perfect matching. We present an algorithm for the assignment problem that uses $SSP(n, m)$ processors, and runs in $O(n^{2/3}\log^3 n(\log nC))$ time if the edge weights are integers in the range $[-C, C]$. Under the assumption that edge weights are given in unary, this problem is known to belong to RNC [77, 102]. Our algorithm is sublinear under this assumption. The previous fastest deterministic algorithm, due to Gabow and Tarjan [45], runs in $\Theta(n\log^3 n\log(nC))$ time in the worst case.

Flows in zero-one networks We study flows in networks with unit capacity arcs (zero-one flow). We consider two versions of the problem, the maximum flow problem and the minimum-cost flow problem. The bipartite matching problem is a special case of the maximum flow problem, and the assignment problem is a special case of the minimum cost flow problem. Our algorithm for maximum flow in a network with unit capacity arcs runs in $O(m^{2/3}\log n)$ time using $BFS(n, m)$ processors. We also show that if the network has no multiple arcs, the algorithm can be modified to run in $O((nm)^{2/5}\log n)$ time. Our algorithm for the minimum cost flow in a network with unit capacity arcs runs in $O(m^{2/3}\log^2 n\log(nC))$ time using $SSP(n, m)$ processors.

The chapter is organized as follows. Section 5.2 presents definitions and notation used throughout the chapter. In Section 5.3 we describe a parallel algorithm for the maximal node-disjoint paths problem and show how to apply it to depth-first search in undirected graphs. In Section 5.4 we give parallel algorithms for maximum matching and zero-one flow problems; in Section 5.5 we extend the results to the weighted versions of these problems.

5.2 Definitions and Notation

This section presents the notation and definitions used throughout the chapter. We assume that the reader is familiar with the standard definitions of maximum flow, minimum cost flow, and maximum matching problems (see, for example, [85] for more details).

Given a graph $G = (V, E)$, let n denote the number of nodes in the graph and let m denote the number of edges (or arcs, if the graph is directed). Throughout the chapter we shall deal only with simple paths, and a path will always mean a simple path. The length of a path is defined as follows. If there is a length associated with each edge, then the length of the path is the sum of the lengths of the edges on the path. Otherwise, the length of the path is the number of edges on the path, *i.e.*, each edge is assumed to have length 1.

A *matching* is a set of edges such that each node in the graph is incident to at most one edge in the matching. A *perfect* matching is a matching such that each node in the graph is incident to exactly one edge in the matching, and a *maximal* matching is a matching such that there is no edge between any two unmatched nodes. The weight of a matching is the sum of the weights of the edges in the matching.

A *pseudoflow* is a function on arcs of the network that obeys capacity constraints. Given a pseudoflow f and a node v, we define the *excess* at v, $e_f(v)$, to be the difference between the incoming and the outgoing flows. A *flow* is a pseudoflow that obeys conservation constraints, *i.e.* excesses at all nodes except the sink and the source are zero.

The *residual capacity* of an arc (v, w) with respect to a pseudoflow f is defined to be the capacity of (v, w) minus $f(v, w)$, and is denoted by $u_f(v, w)$. Given a pseudoflow f, we denote the corresponding residual graph by $G_f = (V, E_f)$, where E_f is the set of arcs with positive residual capacity.

5.3 Maximal Node-Disjoint Paths

This section presents an efficient parallel algorithm that finds a maximal set of node-disjoint paths from a set of sources to a set of sinks in a directed or an undirected graph. This algorithm, which was the starting point of the research described in this chapter, is due to Vaidya. We describe the variation of the algorithm that works for undirected graphs. The extension to the directed case is straightforward.

A natural approach to solve this problem is to find paths one by one. The problem with this approach is that there can be a large ($\Omega(n)$) number of paths, which leads to a running time that is at least linear. Another approach is to maintain a current set of paths, extending as many paths as possible at each iteration. This approach has two problems. First, it takes time that is proportional to the length of the longest path, and therefore it is slow if the paths are long. Second, it may not be possible to extend many paths at each iteration because of the interaction among the paths. By combining these approaches, however, we can achieve a good running time.

The algorithm MAXIMAL-PATHS solves a slight generalization of the node-disjoint paths problem: Given a set $\mathcal{P}_{in}$ of node-disjoint paths connecting sources to intermediate nodes, find a set $\mathcal{P}_{out}$ of node-disjoint paths from the sources to the sinks, such that for any node that is on a path in $\mathcal{P}_{in}$ but not on any path in $\mathcal{P}_{out}$, every path from this node to a sink intersects a path in $\mathcal{P}_{out}$. The node-disjoint paths problem corresponds to the case when each one of the input paths is a single source node. The generalization of the problem is required for the depth-first search algorithm which is described at the end of this section.

Figure 5.1 describes the MAXIMAL-PATHS algorithm. The algorithm maintains two sets of node-disjoint paths: *Active* paths and *Dead* paths, denoted by $\mathcal{P}_a$ and $\mathcal{P}_d$, respectively. An active path starts at a source and ends at some intermediate node which is not a sink; a dead path connects a source to a sink. The initial set of active paths $\mathcal{P}_a$ is the set of the input paths $\mathcal{P}_{in}$. The nodes are divided into *idle, active*, and *dead*, denoted by V_I, V_a, and V_d, respectively. A node is *active* if it belongs to a path, *dead* if it was active during the algorithm but currently does not belong to any path, and *idle* otherwise. Intuitively, a node becomes *dead* if the current set of active paths can be extended to a maximal set of node-disjoint paths

```
procedure MAXIMAL-PATHS($V, E, \mathcal{P}_a$);
    $\mathcal{P}_a$   – the set of active paths;
    $V_I$   – the set of idle nodes;
    $V_a$   – the set of active nodes;
    $V_d$   – the set of dead nodes;
    $T$     – the set of sink nodes;
{The first stage}
    $V_a \leftarrow$ nodes on paths in $\mathcal{P}_a$;
    $V_I \leftarrow V - V_a$;
    $V_d \leftarrow \emptyset$;
    while $|\mathcal{P}_a| \geq \sqrt{n}$ do begin
        $H \leftarrow$ set of end-points of paths in $\mathcal{P}$;
        $H' \leftarrow \{v' : v' \in V_I, \exists v \in H \text{ s.t. } (v, v') \in E\}$;
        $M \leftarrow$ maximal matching on $(H \times H') \cap E$;
        for all $(v, v') \in M$ do begin
            extend the path corresponding to $v$ with $v'$;
            $V_I \leftarrow V_I - v'$;
            $V_a \leftarrow V_a + v'$;
            if $v' \in T$, remove this path from the set of active paths $\mathcal{P}_a$;
        end;
        for all $v \in H$ not matched in $M$ do begin
            remove $v$ from its path;
            if no nodes left on this path, remove it from $\mathcal{P}_a$;
            $V_a \leftarrow V_a - v$;
            $V_d \leftarrow V_d + v$;
        end;
    end;
{The second stage - number of active paths is below $\sqrt{n}$}
    for all $P \in \mathcal{P}_a$ do begin
        $E' \leftarrow ((V_I \times V_I) \cup (V_a \times V_I)) \cap E$;
        $v_r \leftarrow$ the node closest to the end of $P$ from which an idle sink is reachable via edges in $E'$;
        remove nodes that follow $v_r$ from $P$ and add them to $V_d$;
        extend $P$ to a sink via edges in $E'$, add used nodes to $V_a$;
        remove $P$ from $\mathcal{P}_a$;
    end;
end.
```

Figure 5.1: The MAXIMAL-PATHS procedure

without using this node. Initially, V_d is empty and V_I is the set of nodes not on any input path.

The algorithm consists of two stages. The first stage proceeds in iterations, where at each iteration the algorithm extends some of the active paths by *idle* nodes, changing the status of these nodes to *active*. The algorithm "clips" the other active paths, *i.e.*, removes end-point nodes from these paths, and changes the status of the removed nodes to *dead*. Let H be the set of nodes that are the end-points of the active paths, and H' be the set of *idle* nodes that are neighbors of nodes in H. First, the algorithm finds a maximal matching in the bipartite graph induced by the set of edges $(H \times H') \cap E$, where E is the set of edges in the input graph. If a node $v \in H$ is matched to $v' \in H'$, then the path associated with v is extended, and v' becomes the new end-point of the path, changing its status to *active*. If v' is one of the sinks, this path changes its status to *dead*. If a node $v \in H$ is not matched, the path associated with v is "clipped": the node previous to v on this path becomes the new end-point of the path, and the status of v changes to *dead*. This stage continues as long as the number of active paths is at least $\sqrt{n}$.

During the second stage, the algorithm extends the active paths to sinks one by one. To extend a path $P = (v_1, v_2, \ldots, v_k)$, the algorithm first computes connected components in the graph induced by edges in $(V_I \times V_I) \cup (V_a \times V_I)$. Let v_r be the node on P, such there exists a path P' from v_r to some sink t via *idle* nodes, and for all $r < i \leq k$, no *idle* sink is reachable from v_i by a path that consists of *idle* nodes only. Then the algorithm clips P, changes the status of the nodes $\{v_i : r < i \leq k\}$ to *dead*, changes the status of nodes on P' to *active*, and extends the path $(v_1, v_2, \ldots, v_r)$ by attaching it to P'. If such node was not found, *i.e.* it is impossible to extend P using idle nodes, all nodes on P are marked *dead*.

By construction, the algorithm terminates with a set of paths where each path connects a source to a sink, and each source or sink belongs to at most one path. Moreover, a source is marked *dead* if and only if it does not belong to any path and a sink is marked *idle* if and only if it does not belong to any path. Thus, the following lemma is sufficient to show correctness of the algorithm.

Lemma 5.3.1 At any moment during an execution of the algorithm, there is no path from a *dead* node to an *idle* sink such that all the nodes on this path are either *dead* o: *idle*.

Proof: Consider an iteration of the first stage in which a node v becomes *dead*. By construction, v becomes *dead* only if it was not matched during computation of the maximal matching. This means that at the end of this iteration v does not have any *idle* neighbors. On the other hand, if a node changed its status to *dead* during the second stage, then, by construction, there is no path consisting of *idle* nodes only from this node to an *idle* sink. Furthermore, a node cannot change its status to *idle* from any other status, and hence each path from a *dead* node to an *idle* sink must pass through an *active* node. ∎

We analyze the running times of the first and the second stages separately. The following lemma bounds the number of iterations in the first stage.

Lemma 5.3.2 There are at most $O(\sqrt{n})$ iterations in the first stage.

Proof: The main idea of the proof is that nodes can change status only "in one direction", and that at each iteration a large number of nodes change status. Define a potential function

$$\Phi = |V_A| + 2|V_I|.$$

An extension of a path by one node changes the status of this node from *idle* to *active* and reduces Φ by 1. On the other hand, when a path is clipped, its old end-point changes the status from *active* to *dead*, again reducing Φ by 1. At each iteration of the first stage there are at least $\sqrt{n}$ active paths. At the end of an iteration each one of these paths is either extended or clipped, which causes a total reduction of at least $\sqrt{n}$ in Φ. The claim follows, because initially $\Phi \leq 3n$. ∎

Each iteration of the first stage can be implemented in $O(\log^3 n)$ time and with $O(m)$ processors, using the maximal matching algorithm of Israeli and Shiloach [72]. Each iteration of the second stage of the algorithm is essentially a connectivity computation, which can be computed in $O(\log n)$ time and $O(m)$ processors for the undirected case [117], and $O(\log^2 n)$ time and $BFS(n,m)$ processors for the directed case [108]. This leads to the following theorem.

Theorem 5.3.3

1. On undirected graphs, the MAXIMAL-PATHS algorithm runs in $O(\sqrt{n}\log^3 n)$ time using $O(n+m)$ processors.
2. On directed graphs, the MAXIMAL-PATHS algorithm runs in $O(\sqrt{n}\log^3 n)$ time using $BFS(n,m)$ processors.

Observe that a maximal set of node-disjoint paths corresponds to a blocking flow in matching networks (described in detail in the next section). Thus, by using the MAXIMAL-PATHS procedure to find blocking flow at each iteration of Dinic's maximum flow algorithm [34, 38], we can compute a maximum bipartite matching in sublinear time. In the subsequent sections we will show more efficient algorithms for bipartite matching and related problems; these algorithms do not use the MAXIMAL-PATHS algorithm.

Depth-First Search Another application of the MAXIMAL-PATHS algorithm is for constructing a deterministic sublinear-time algorithm for finding a depth-first search tree in an undirected graph. The problem of finding such a tree has been studied before [47, 121], and recently Aggarwal and Anderson have found a *randomized* NC algorithm for it [6]. However, no deterministic sublinear-time parallel algorithm for the problem was known previously.

Although the Aggarwal-Anderson algorithm is randomized, the randomization is used only in order to compute a *maximum* set of node-disjoint paths with the minimum weight [6]. Aggarwal and Anderson reduce this problem to the assignment problem. A careful examination of their proofs shows that instead of a *maximum* set of paths, it is sufficient to be able to find a *maximal* set of paths. More precisely, it is sufficient to have an algorithm that solves exactly the generalization of the node-disjoint path problem that is solved by the MAXIMAL-PATHS algorithm. Therefore, we have the following theorem.

Theorem 5.3.4 A depth-first search tree in an undirected graph can be found in $O(\sqrt{n}\log^5 n)$ time using $O(n+m)$ processors.

The proof of the theorem involves a straightforward combination of the Aggarwal-Anderson algorithm [6] and the MAXIMAL-PATHS algorithm.

5.4 Bipartite Matching and Zero-One Flows

In this section we describe sublinear-time parallel algorithms for the bipartite matching problem and for the zero-one network flow problem (both with and without multiple arcs). This section consists of three parts. In the first part we review Goldberg-Tarjan's generic maximum flow algorithm. In the second part we present our maximum matching algorithm, and in the third part we present our algorithm for finding a maximum flow in networks with unit capacities.

5.4.1 Goldberg-Tarjan Maximum Flow Algorithm

In this section we review the Goldberg-Tarjan push/relabel framework for solving the maximum flow problem [59, 62]. We present only those ideas which are relevant for finding flows in unit-capacity networks.

Before describing the generic Goldberg-Tarjan algorithm, we need to introduce a few terms. For more detailed definitions, see [62]. A (valid) *distance labeling* is an integer-valued function d on nodes that satisfies $d(v) \leq d(w) + 1$ for every residual arc (v, w). Given a pseudoflow f and a distance labeling d, we define

$$E(f,d) = \{(v,w) \in E_f | d(v) = d(w) + 1\}$$

to be the the set of *admissible* arcs; the *admissible* graph $G(f,d) = (V, E(f,d))$ is the graph induced by the arcs in $E(f,d)$.

The algorithm is based on PUSH and RELABEL operations described in Figure 5.2. PUSH moves one unit of excess through an admissible arc; RELABEL changes the distance labeling d of a node to create an outgoing admissible arc from this node while maintaining the validity of the distance labeling.

The generic Goldberg-Tarjan maximum flow algorithm is shown in Figure 5.3. First we set all distance labels to 0, except the label at the source, which is set to n. Then we saturate all

Push(v, w).
Applicability: $e_f(v) > 0$, $u_f(v,w) > 0$ and $d(v) = d(w) + 1$.
Action: Send $\delta = \min(e_f(v), u_f(v,w))$ units of flow from v to w as follows:
$f(v,w) \leftarrow f(v,w) + \delta$; $f(w,v) \leftarrow f(w,v) - \delta$;
$e_f(v) \leftarrow e_f(v) - \delta$; $e_f(w) \leftarrow e_f(w) + \delta$.

Relabel(v).
Applicability: Any v.
Action: $d(v) \leftarrow \min\{d(w) + 1 | (v,w) \in E_f\}$.
(If this minimum is over an empty set, $d(v) \leftarrow \infty$.)

Figure 5.2: Push and relabel operations.

arcs that emanate from the source. Note that at this point the distance labeling becomes valid. Now we apply Push and Relabel operations in any order until no more excesses are left.

Goldberg and Tarjan [62] showed that the Generic-Max-Flow algorithm terminates with a valid maximum flow. Observe, that for the case of zero-one flows, an arc can participate in at most one Push operations before one of its ends is relabeled. Since labels can not grow beyond $2n$, this leads to $O(nm)$ bound on the number of Push operations, and $O(n^2)$ number of Relabel operations. Goldberg [58] gave a parallel implementation of the generic algorithm which runs in $O(n^2 \log n)$ time.

5.4.2 Bipartite Matching Algorithm

To solve a bipartite matching problem, we transform it into a zero-one network flow problem in a standard way (see, for example, [85]). Given a bipartite graph with a node set $S \bigcup T$, we direct edges of the graph from nodes in S to nodes in T. We add a source s and arcs (s, v) for all $v \in S$, and a sink t and arcs (w, t) for all $w \in T$. We define all arc capacities to be one. The resulting maximum flow problem is equivalent to the original bipartite matching problem. The network which can be obtained by the above transformation is called a *matching network*. We denote vertices and edges of the matching network by V and E, respectively.

Two possible approaches to the design of parallel algorithms for the problem of finding maximum flows in matching networks suggest themselves. One approach is to use the Ford-Fulkerson

```
procedure GENERIC-MAX-FLOW($V, E, s, t$);

    [first stage – initialization]
    for all $v \in V - \{s\}$ do $d(v) \leftarrow 0$;
    $d(s) \leftarrow n$;
    for all $(v, w) \in E$ do $f(v, w) \leftarrow 0$;
    for all $v \in V$ do $e_f(v) \leftarrow 0$;
    for all $v \in V$ such that $(s, v) \in E$ do begin
        $f(s, v) \leftarrow 1$;
        $e_f(v) \leftarrow e_f(v) + 1$;
    end;

    [second stage]
    while there exists a node with an excess do
        apply PUSH or RELABEL in arbitrary order.

    return the resulting flow $f$;
end.
```

Figure 5.3: The generic Goldberg-Tarjan maximum flow algorithm

augmenting path algorithm [41] with a parallel breadth-first search subroutine. Another approach is to use a variant of Goldberg's parallel maximum flow algorithm [58]. Both approaches lead to superlinear-time algorithms, but for different reasons. The bottleneck of the first approach is a potentially large number of augmenting paths; the bottleneck of the second approach is a potentially large number of node relabelings.

Our algorithm works in two stages, using the Goldberg's approach in the first stage and the Ford-Fulkerson approach in the second stage. A proper balancing of the two stages which is achieved by adjusting the *activity parameter* and the *distance parameter* leads to a sublinear running time. In the context of sequential algorithms for the problem, similar balancing, which involves a single parameter that is similar to our distance parameter, was introduced by Even and Tarjan [38] to obtain $O(\sqrt{n}m)$ time bounds.

Intuitively, the key idea of our bipartite matching algorithm is to keep at most a single unit of excess at any node and to use maximal matching to decide where to push the excesses at each parallel step. Essentially, we replace the "while" loop of the generic Goldberg-Tarjan maximum flow algorithm (see Figure 5.3) with the MATCH-AND-PUSH procedure (see Figure 5.5), which, as we will prove below, causes many relabelings to happen at each parallel step.

```
procedure B-MATCH($S, T, E$);

    [initialization]
    transform the input problem into network flow form;

    [first stage]
    for all $v \in S \bigcup T$ do $d(v) \leftarrow 0$;
    $d(t) \leftarrow 0$; $d(s) \leftarrow n$;
    for all $(v, w) \in E$ do $f(v, w) \leftarrow 0$;
    for all $v \in S$ do $f(s, v) \leftarrow 1$;
    for all $w \in T \bigcup \{s, t\}$ do $e_f(w) \leftarrow 0$;
    for all $v \in S$ do $e_f(v) \leftarrow 1$;
    while the number of active nodes is at least $l$ do MATCH-AND-PUSH;
    return all excesses to the source;


    [second stage]
    while there is an augmenting path from $s$ to $t$ do
        find an augmenting path and augment;

    return the matching corresponding to the current (maximum) flow;
end.
```

Figure 5.4: High-level description of the bipartite matching algorithm. For the algorithm described in this chapter, we take $l = n^{2/3}$.

Figure 5.4 describes the algorithm B-MATCH which finds a maximum bipartite matching. The algorithm consists of two stages. The first stage is executed as long as the number of *active* nodes is large, where a node is considered active if it has an excess and its distance label d is below k, the *distance parameter*. By "large" we mean that the number of active nodes is above l, the *activity parameter*. When the number of active nodes is small (below l), excesses from these nodes are returned to the source, and the second stage begins. In this stage, the algorithm finds augmenting paths from the source to the sink one by one, like it is done in the Even-Tarjan maximum matching algorithm [38]. We will prove that the second stage works fast because the residual flow is small.

The first stage of the algorithm starts by initializing the flow to zero, setting distance labels of nodes in $S \bigcup T \bigcup \{t\}$ to zero, and setting the distance label of the source to n. (Throughout the algorithm, distance labels of source and sink never change: $d(s) = n, d(t) = 0$.) Then, all arcs going out of the source are saturated. At this point, all nodes of S have excess of 1. After the initialization is complete, the MATCH-AND-PUSH procedure is executed until the number of active nodes becomes less than l. Recall that in the context of this algorithm, we have defined an active node to be a node with label d below the distance parameter k. (As we shall see, the best running time is achieved for distance parameter $k = \lfloor n^{1/3} \rfloor$ and activity parameter $l = \lfloor \sqrt{nk} \rfloor$.) Finally, at the end of the first stage, the flow excesses are returned to the source. Namely, the excess flow is pushed from nodes $v \in S$ such that $e_f(v) = 1$ to s along (v, s).

The MATCH-AND-PUSH procedure, shown in Figure 5.5, is the key to the first stage of the algorithm. The following lemma states the properties of this procedure that are essential for the analysis of the algorithm.

Lemma 5.4.1

The MATCH-AND-PUSH procedure maintains the following invariants:

1. The current pseudoflow f is integral.
2. Indegree of a node $v \in S$ in the residual graph G_f is $1 - e_f(v)$.
3. For every node $v \in S \bigcup T$, $e_f(v) \in \{0, 1\}$.
4. On entry to and on exit from MATCH-AND-PUSH, all nodes in T have zero excesses.

Proof: Integrality of f follows by induction on the number of the PUSH operations. The second invariant follows from the properties of matching networks.

Invariant 3 holds after initialization by the structure of a matching network. Suppose that the invariant holds before an execution of MATCH-AND-PUSH. Step 1 assures that it holds after Step 2. The relabeling steps 3, 5, and 6 cannot affect this invariant. Because of the second invariant, no flow can be pushed to an active node and at most one unit of flow can be pushed to an inactive node at Step 4. Thus Step 4 preserves the invariant.

Invariant 4 holds because after Step 2, every node in T has excess of either zero or one, and because of the relabeling done at Step 3 every node with excess of one has an outgoing admissible arc that can be used to push the excess from the node in Step 4. After Step 4 all

Step 1. Find a maximal matching in the subgraph of the admissible graph induced by nodes in T and active nodes in S.

Step 2. For every matched arc (v, w), push excess flow from v to w.

Step 3. Relabel nodes in T.

Step 4. Push flow from active nodes in T along admissible arcs.

Step 5. Relabel nodes in T.

Step 6. Relabel nodes in S.

Figure 5.5: The *Match-and-Push* procedure

nodes in T have zero excesses. The remaining steps do not change the pseudoflow, and therefore Invariant 4 holds at the end of Match-and-Push. ▮

From Invariants 3 and 4 in Lemma 5.4.1 it follows that all matched arcs in Step 2 of Match-and-Push are directed from S to T. Lemma 5.4.1 also implies that the last step of the first stage, namely returning flow from the nodes with excess to the source, is easy. More precisely, Invariants 3 and 4 imply that nodes in T have no excesses, and each node in S has at most one unit of excess. Furthermore, since $k \leq n$, no flow is pushed from a node $v \in S$ to s by the previous part of the algorithm, and therefore for all $v \in S$, the residual capacities of arcs (v, s) are equal to one. Thus, the excesses can be pushed from nodes $v \in S$ such that $e_f(v) = 1$ to s along (v, s). Note that these pushes are nonstandard, *i.e.*, they do not preserve the validity of d. This is not a problem, however, because we do not use the distance labels after the last execution of Match-and-Push.

We start the analysis of the algorithm by bounding the running time of the Match-and-Push procedure.

Lemma 5.4.2 Procedure Match-and-Push runs in $O(\log^3 n)$ time on a CRCW PRAM with $n+m$ processors.

Proof: Steps 2-6 can be implemented so that each step takes $O(\log n)$ time on a CRCW PRAM with $n + m$ processors [58, 118]. The bottleneck is Step 1, which takes $O(\log^3 n)$ time on a CRCW PRAM using $n + m$ processors [72]. ▮

The next lemma, which bounds the number of executions of Match-and-Push, is the key to the analysis of the algorithm.

Lemma 5.4.3 Procedure MATCH-AND-PUSH with activity parameter l and distance parameter k is executed at most $\frac{n(k+1)}{l} + 1$ times.

Proof: We show that at all execution of MATCH-AND-PUSH there are at least l relabelings of nodes that have distance label below k at the beginning of the execution. Since the distance labels never decrease, the total number of such relabelings is at most $n(k+1)$, and the desired bound follows.

We claim that a relabeling of each of the following nodes occurs during an execution of MATCH-AND-PUSH:

1. The nodes in T which are matched in Step 1.
2. The active nodes in S which are not matched in Step 1.

Note that in every execution of MATCH-AND-PUSH, except perhaps the last, the number of nodes satisfying these two conditions is at least l, so establishing this claim completes the proof of the theorem.

Suppose a node $w \in T$ is matched with a node $v \in S$ at Step 1, which implies that $d(v) \leq k$ and $d(w) = d(v) - 1 < k$. We show that $d(w)$ increases at Step 3 or at Step 5. If $d(w)$ increases at Step 3, we are done.

Consider a case in which $d(w)$ did not increase at Step 3. After Step 4, the only residual arc out of w is (w, v). (This follows from the observation that the outdegree of a node $w \in T$ in the residual graph G_f is $1 + e_f(w)$ and that after Step 4 we have $e_f(w) = 0$.) At Step 1, the arc (v, w) is admissible, and therefore $d(v) = d(w) + 1$. Node v has not been relabeled since then, so $d(v)$ did not change, and we have assumed that neither had $d(w)$. By the definition of the relabeling operation, at Step 5 the distance label of w becomes $d(v) + 1$, *i.e.*, the distance label increases by 2.

Now consider an active node $v \in S$ that is not matched at Step 1. Recall that by definition of an active node, $d(v) \leq k$, and thus the arc (v, s) cannot be admissible, because $d(s) = n$

and $d(v) \leq k \leq n$. Hence, in the beginning of Step 1 all admissible neighbors of v lie in T. These neighbors are matched during this step, and therefore by Step 6 their distance labels must increase (by the argument above). Since v has not acquired any new residual neighbors, its distance label must increase at Step 6. ▮

Lemmas 5.4.2 and 5.4.3, combined with an observation that the initialization of the first stage can be done in constant time using $n + m$ processors, imply the following result.

Lemma 5.4.4 The first stage of the bipartite matching algorithm runs in $O(\frac{nk}{l} \log^3 n)$ time using $n + m$ processors.

To complete the analysis of the algorithm, we prove the following lemma, which is similar to a lemma in Even and Tarjan [38].

Lemma 5.4.5 After the first stage of the algorithm, the value of the residual flow is at most $n/k + l$, where k and l are distance and activity parameters, respectively.

Proof: Since returning excesses to the source does not affect the amount of flow that can reach the sink, it suffices to show that the amount of flow that can reach the sink after the last execution of MATCH-AND-PUSH is at most $n/k + l$. The proof uses the fact that for every node v, the label $d(v)$ is a lower bound on the distance in the residual graph from v to the sink [58].

Consider the pseudoflow f and the distance labeling d just after the last execution of MATCH-AND-PUSH, and let $\overline{f}$ be an optimal flow. Consider the set of arcs $A = \{(i,j) | \overline{f}(i,j) > f(i,j)\}$. Note that $A \subseteq E_f$. Arcs in A can be partitioned into a collection of simple paths from nodes with excess to t, a collection of simple paths from nodes with excess to s, and a collection of cycles. Even and Tarjan showed that by the properties of matching networks these paths and cycles are node-disjoint [38].

We need to show that the number of paths in the first collection is at most $l + n/k$. Consider a residual path from a node v to t. Since $d(v)$ is a lower bound on the distance from v to t in G_f, the length of such a path is at least $d(v)$. Thus, at most l paths in the first collection have length of k or less. The remaining paths have length greater than k, and the number of such paths is at most n/k, because the paths are node-disjoint. ▮

In the second stage of the algorithm, we find augmenting paths one by one. This stage terminates when there are no augmenting paths, and therefore produces correct result. The maximum number of augmenting paths found during this stage is bounded by Lemma 5.4.5, and therefore we have the following claim.

Lemma 5.4.6 The second stage of the bipartite matching algorithm runs in $O((\frac{n}{k}+l)\log^3 n)$ time using $BFS(n,m)$ processors, where k and l are the distance and the activity parameters, respectively.

Remark: In the second stage, the algorithm can use any augmenting path. The fastest current parallel algorithm, however, finds a shortest augmenting path.

Theorem 5.4.7 *The bipartite matching algorithm runs in* $O(n^{2/3}\log^3 n)$ *time using* $BFS(n,m)$ *processors.*

Proof: Set $k = \lfloor n^{1/3} \rfloor$ and $l = \lfloor n^{2/3} \rfloor$ and apply Lemmas 5.4.4 and 5.4.6. ▮

5.4.3 Zero-One Flow Algorithms

In this section we describe algorithms for computing maximum flows in networks with unit arc capacities. We describe two algorithms, one optimized for general zero-one networks and another optimized for networks with no multiple arcs. For general zero-one networks, by transforming the input network into a matching network [25, 77] and applying the bipartite matching algorithm described in the previous section, it is possible to achieve a running time which is only $\log^2 n$-factor worse than the running time of the algorithm described below. However, the method we describe in this section leads to better bounds for networks with no multiple arcs.

First we describe the algorithm that finds a maximum flow in a general zero-one network (V, E, s, t) with unit arc capacities (see Figure 5.6). At a high level, this algorithm is similar to the algorithm of the previous section. The algorithm consists of two stages, where the first one is based on Goldberg's parallel maximum-flow algorithm, and the second based on the Ford-Fulkerson method. The balancing of work done in the two stages is similar to that of the sequential algorithms of Even and Tarjan [38] and, more recently, Ahuja and Orlin [8]. In addition, the zero-one flow algorithm has a finish-up stage that converts a pseudoflow of

```
procedure ZERO-ONE($V, E, s, t$);

    [first stage]
    for all $v \in V - \{s\}$ do $d(v) \leftarrow 0$;
    $d(s) \leftarrow n$;
    for all $(v, w) \in E$ do $f(v, w) \leftarrow 0$;
    for all $v \in V$ do $e_f(v) \leftarrow 0$;
    for all $v \in V$ such that $(s, v) \in E$ do begin
        $f(s, v) \leftarrow 1$;
        $e_f(v) \leftarrow e_f(v) + 1$;
    end;
    while the total amount of excess at active nodes is at least $l$ do PUSH-AND-RELABEL;

    [second stage]
    while there is an augmenting path from a node $v \in V - \{s, t\}$ such that $e(v) > 0$ to $t$ do
        find an augmenting path from $v$ to $t$ and augment;

    [finish-up stage]
    if the current pseudoflow $f_{end}$ is not a flow,
        convert it into a flow by recursively calling ZERO-ONE;
    return($\overline{f}$);
end.
```

Figure 5.6: **High-level description of the zero-one flow algorithm. For general zero-one networks, take $l = m^{2/3}$; for zero-one networks with no multiple arcs, take $l = \min(m^{2/3}, (nm)^{2/5})$.**

maximum value with only positive excesses into a flow of maximum value, where the value of a pseudoflow is the amount flowing into the sink.

Intuitively, the key idea is to associate "energy" with every unit of excess, which is equal to the amount of possible future increase in the distance label of the node this unit of excess resides at. The total energy of a pseudoflow is bounded by $O(nm)$. We use this fact to show that during a parallel step either much work is done, or much energy is used.

The first stage of the algorithm is essentially Goldberg's parallel maximum flow algorithm [59, 58, 62] with two important modifications. First, we define an active node as a node that has a distance label of k or less, where k is the *distance parameter*. The second modification is that the stage terminates when the total amount of excess at active nodes is less then $l = m^{2/3}$, where l is the *activity parameter*. The key part of this stage is the PUSH-AND-RELABEL procedure. This procedure can be implemented by using either techniques of Shiloach and Vishkin [117] or by using parallel prefix computations (see *e.g.* [19, 87]) as described by

```
Step 1. For all active nodes $v$, use a parallel prefix computation on the list of outgoing arcs to distribute
        $e_f(v)$ among neighbors of $v$ in the admissible graph.
Step 2. For all nodes $v$, use parallel prefix computation on the list of incoming arcs to compute new
        excess $e_f(v)$ by adding up flow pushed to $v$ during Step 1.
Step 3. Using parallel prefix computations, relabel all nodes $v \neq s, t$.
```

Figure 5.7: The *Push-and-Relabel* procedure.

Goldberg [58]. Figure 5.7 describes the procedure. (Note that Step 3 of Push-and-Relabel can be replaced by a computation of true distances from all vertices to the sink in the residual graph, which can be achieved by doing a breadth-first search backwards from the sink.)

The second stage of the algorithm repeatedly finds an augmenting path from a node $v \notin \{s,t\}$ with $e_f(v) > 0$ to the sink, and augments along the path. One way to find an augmenting path is to do a breadth-first search backwards from the sink in the residual graph. Observe, that when no such paths exist, the current pseudoflow f_{end} is of maximum value. Moreover, since we have not created any negative excesses, all excesses are positive.

The finish-up stage converts f_{end} into a flow $\overline{f}$ by returning excesses from nodes in $V - \{s,t\}$ to s. This conversion is done by running the same algorithm on a modified network. The modified network is obtained from $G_{f_{end}}$ by adding a new source s' and new arcs of unit capacity connecting s' to nodes in $V - \{s,t\}$ that have excesses with respect to f_{end}. If a node v has excess $e_{f_{end}}(v)$, then $e_{f_{end}}(v)$ arcs of the form (s',v) are added. The source of the original network is the sink of the modified network. It can be easily shown (see [59, 62]) that when the zero-one flow algorithm is applied to the modified network, its second stage terminates with a flow (rather than a pseudoflow).

The correctness of the algorithm can be shown in exactly the same way as it was done by Goldberg [58]. Performance of the Push-and-Relabel procedure is summarized by the following lemma, which follows from the results of Blelloch [19] and Goldberg [58].

Lemma 5.4.8 Procedure Push-and-Relabel runs in $O(\log n)$ time using m processors.

The next lemma, which is the key to the analysis of this algorithm, bounds the number of times Push-and-Relabel is executed.

Lemma 5.4.9 Procedure PUSH-AND-RELABEL is executed $O(\frac{mk}{l})$ times, where k and l are distance and activity parameters, respectively.

Proof: Define potential function Φ to be equal to the number of pushes plus the sum over all nodes $v \in V$ of the label $d(v)$ times the degree of v, where by "degree" we mean the sum of indegree and outdegree.

The pushes are made only through admissible arcs, which means that if a push was made from v to w, the next push associated with this edge has to be from w to v, which can happen only after the label of w was increased by at least 2. The algorithm executes pushes only from nodes with label of at most k, and hence the total number of pushes is $O(mk)$. Therefore Φ is bounded by $O(mk)$ as well.

Consider a unit of excess at an active node v at the beginning of the PUSH-AND-RELABEL procedure. If this unit participated in a push at Step 1, it increased Φ by one. Otherwise, the label of v increased by at least 1 during Step 3. This means that Φ was increased by at least the degree of v during this step. Observe, that each unit of excess at v can be associated with the arc over which it reached v, and hence the degree of v is at least equal to the number of units of excess at v. During the first stage the total excess at active nodes is at least l, which means that each call to PUSH-AND-RELABEL increases Φ by at least l, and the bound follows. ∎

Lemmas 5.4.8 and 5.4.9, combined with an observation that the initialization of the first stage can be done in constant time using $n + m$ processors, imply the following result.

Lemma 5.4.10 The first stage of the zero-one flow algorithm runs in $O(\frac{mk}{l} \log n)$ time using $n + m$ processors, where k and l are distance and activity parameters, respectively.

In the second stage we find augmenting paths one by one. To bound the running time of the stage, we first bound the value of the residual flow after the execution of the first stage. The following lemma is similar to Lemma 5.4.3.

Lemma 5.4.11 After the first stage of the algorithm is applied to a general zero-one network, the value of the residual flow is at most $m/k + l$.

Proof: The proof of this lemma is exactly like the proof of Lemma 5.4.5. The only difference is that we decompose the residual flow into arc-disjoint paths instead of node-disjoint paths. ▮

Lemma 5.4.12 The second stage of the zero-one flow algorithm runs in $O((\frac{m}{k} + l)\log n)$ time using $BFS(n, m)$ processors, where k and l are distance and activity parameters, respectively.

Proof: The lemma follows from Lemma 5.4.11. ▮

Theorem 5.4.13 On general zero-one networks, the zero-one flow algorithm runs in $O(m^{2/3}\log n)$ time using $BFS(n, m)$ processors.

Proof: Set $k = \lfloor m^{1/3} \rfloor$ and $l = \lfloor m^{2/3} \rfloor$. Lemmas 5.4.10 and 5.4.12 imply that the first two stages of the algorithm run in the desired resource bound. These lemmas also imply that the finish-up stage runs in the same resource bounds. ▮

Now we consider the problem of finding maximum flows in zero-one networks with no multiple arcs. In this case, we can improve the time bound of Theorem 5.4.13 for dense graphs (more precisely, for $m > n^{3/2}$). The following lemma, which is an adaptation of a similar lemma due to Even and Tarjan [38], is a key to the improvement.

Lemma 5.4.14 After the first stage of the algorithm is applied to a zero-one network with no multiple arcs, the value of the residual flow is at most $\left(\frac{2n}{k-1}\right)^2 + l$, where k and l are distance and activity parameters, respectively.

Proof: Since returning excesses to the source does not affect amount of flow that can reach the sink, it suffices to show that the amount of flow that can reach the sink after the last execution of PUSH-AND-RELABEL is at most $\left(\frac{2n}{k-1}\right)^2 + l$.

Consider the pseudoflow f and the distance labeling d just after the last execution of MATCH-AND-PUSH, and let $\overline{f}$ be an optimal flow. Consider the set of arcs $A = \{(i,j) | \overline{f}(i,j) > f(i,j)\}$. Note that $A \subseteq E_f$. Arcs in A can be partitioned into a collection of simple paths from nodes with excess to t, a collection of simple paths from nodes with excess to s, and a collection of cycles. Since we have a zero-one network, these paths and cycles are arc-disjoint. The number of paths in the first collection that start at a node with a distance label of k or less is at most l.

To complete the proof, we need to show that the number of paths that start at a node with a distance label greater then k and reach the sink is at most $\left(\frac{2n}{k-1}\right)^2$. Suppose for contradiction that this is false. Let P be the set of these paths, and let $G' = (V, E')$ be the graph induced by arcs on paths in P. Let $d'(v)$ be the distance in G' from v to t. Let $V_i = \{v \in V | d'(v) = i\}$. By definition of P, no path in P starts at a node in the set $V_0 \bigcup V_1 \bigcup \ldots \bigcup V_k$. Therefore $|P|$ is bounded, for $0 \leq j \leq k-1$, by the number of arcs in the set $E \bigcap (V_j \times V_{j+1})$, which is at most $|V_j| \times |V_{j+1}|$, since the network has no multiple arcs. Our assumption implies that $|V_j| \times |V_{j+1}| > \left(\frac{2n}{k-1}\right)^2$, and therefore $|V_j| + |V_{j+1}| > \frac{2n}{k-1}$, for $0 \leq j \leq k-1$. We obtain a contradiction as follows:

$$\begin{aligned}
& (V_0 + V_1) + (V_2 + V_3) + \ldots + (V_{2\lfloor k/2 \rfloor - 1} + V_{2\lfloor k/2 \rfloor}) \\
> \; & \tfrac{2n}{k-1} \lfloor \tfrac{k}{2} \rfloor \\
> \; & \tfrac{2n}{k-1} \tfrac{k-1}{2} \\
> \; & n.
\end{aligned}$$

▮

Using lemmas 5.4.11 and 5.4.14, one can obtain the following theorem; the proof is similar to the proof of Theorem 5.4.13.

Theorem 5.4.15 **On zero-one capacity networks with no multiple arcs, the maximum flow algorithm** *runs in time* $O(min(m^{2/3}, (nm)^{2/5}) log\, n)$ *on a* **CRCW PRAM** *using* $BFS(n, m)$ **processors.**

5.5 The Assignment Problem and Minimum-Cost Flows

In this section we describe parallel algorithms for the weighted versions of the problems studied in the previous section, namely the assignment problem and the minimum-cost flow problem with unit capacities. The section consists of three parts. In the first part we present some definitions and review the generic minimum-cost flow algorithm of Goldberg and Tarjan [58, 64]. In the second part we present our parallel algorithm for weighted bipartite matching, and in the third part we discuss the problem of finding a minimum-cost flow in a general network with unit capacities. For the purpose of this section, we assume familiarity with the minimum-cost flow framework developed by Goldberg and Tarjan [58, 61, 64].

```
procedure MIN-COST-FLOW($V, E, c$);

    $C \leftarrow \max\{|c(v,w)| : (v,w) \in E\}$;
    $\epsilon \leftarrow C$;
    for all $v \in V$ do $p(v) \leftarrow 0$;
    $p(s) = -2nC$;

    while $\epsilon \geq 1/n$ do
        $\epsilon \leftarrow \epsilon/2$;
        $(c,p,f) \leftarrow$ GENERIC-REFINE($V, E, c, p, f, \epsilon$);
    end;
    return the current (maximum) flow $f$;
end.
```

Figure 5.8: **High-level description of the outer (scaling) loop of the generic Goldberg-Tarjan minimum-cost flow algorithm.**

5.5.1 Goldberg-Tarjan Minimum-Cost Flow Algorithm

In this section we review the main ideas of the minimum-cost flow framework developed by Goldberg and Tarjan [58, 61, 64]. We review only those ideas that are relevant for networks with unit capacities.

In order to describe the algorithm, we need to introduce a few definitions (see [58, 61] for more details). Each node v is assigned a *price* $p(v)$. Given p, the *reduced cost* of an arc (v,w) is defined by

$$c_p(v,w) = p(v) - p(w) + c(v,w),$$

where $c(v,w)$ is the original cost that is part of the input to the problem. We say that a pseudoflow is *ϵ-optimal* if there are no residual arcs with reduced cost below $-\epsilon$. Given a pseudoflow f and a price function p, an arc $(v,w) \in E$ is *admissible* if it is a residual arc with negative reduced cost, *i.e.*, if $u_f(v,w) > 0$ and $c_p(v,w) < 0$. The *admissible graph* is the graph induced by the set of admissible arcs. Define $c(f)$, the *cost* of pseudoflow f, by

$$\sum_{(v,w)\in E:f(v,w)>0} c(v,w)f(v,w).$$

In the case of *zero-one flows*, the cost of a pseudoflow is equal to the sum of the costs of the saturated arcs.

```
procedure GENERIC-REFINE($V, E, c, p, f, \epsilon$);

    [Convert into $\epsilon$-optimal pseudoflow.]

    for all $\{(v,w) \mid (v,w) \in E_f$ and $c_p(v,w) < -\epsilon\}$ do
        $f(v,w) \leftarrow f(v,w) + 1$;
        $e_f(w) \leftarrow e_f(w) + 1$;
        $e_f(v) \leftarrow e_f(v) - 1$;
    end;

    [convert into $\epsilon$-optimal flow.]

    while there exists a node with excess do
            apply PUSH and RELABEL in arbitrary order;

    return $(c, p, f)$;

end.
```

Figure 5.9: High-level description of the inner loop of the generic Goldberg-Tarjan minimum-cost flow algorithm.

The outer loop of the minimum-cost flow algorithm, shown in Figure 5.8, does generalized cost-scaling [58, 61]. Initially $\epsilon = C$, where C is the maximum absolute value of an input cost. The algorithm iteratively halves ϵ and uses the GENERIC-REFINE procedure (see Figure 5.9) to update the flow to be ϵ-optimal again. Bertsekas [18] noticed that if the costs are integral then an ϵ-optimal flow is optimal for $\epsilon < 1/n$, and therefore we have the following lemma.

Lemma 5.5.1 (Goldberg and Tarjan [58, 61]) The algorithm terminates and produces an optimal flow after $O(\log nC)$ calls to the GENERIC-REFINE procedure.

The heart of the algorithm is the GENERIC-REFINE procedure, shown in Figure 5.9, that converts a 2ϵ-optimal flow into an ϵ-optimal flow. The procedure is similar to the maximum flow algorithm which we reviewed in Section 5.4.1. GENERIC-REFINE starts with constructing an ϵ-optimal pseudoflow by saturating residual arcs with reduced cost of below $-\epsilon$ and creating appropriate excesses and deficits at the nodes. Then positive excesses are moved towards negative ones using PUSH and RELABEL procedures, described in Figure 5.10. Similar to the case of the generic maximum flow, PUSH moves one unit of excess through an admissible arc (*i.e.* an arc with negative reduced cost); RELABEL changes *the price p of a node to create an*

```
PUSH($v, w$).
Applicability: $e(v) > 0$, $u_f(v,w) > 0$ and $c_p(v,w) = p(v) - p(w) + c(v,w) \leq 0$.
Action:        Send $\delta = \min(e_f(v), u_f(v,w))$ units of flow from $v$ to $w$ as follows:
               $f(v,w) \leftarrow f(v,w) + \delta$; $f(w,v) \leftarrow f(w,v) - \delta$;
               $e_f(v) \leftarrow e_f(v) - \delta$; $e_f(w) \leftarrow e_f(w) + \delta$.

RELABEL($v$).
Applicability: Any $v$.
Action:        $p(v) \leftarrow \max\{p(w) - c(v,w) - \epsilon | (v,w) \in E_f\}$.
               (If this maximum is over an empty set, $p(v) \leftarrow -\infty$.)
```

Figure 5.10: **Push and relabel operations for minimum-cost flow computation.**

outgoing admissible arc from this node while maintaining ϵ-optimality.

Goldberg and Tarjan proved that the MIN-COST-FLOW algorithm produces a feasible minimum-cost flow of maximum value upon termination. Moreover, they showed that during a single execution of GENERIC-REFINE the amount of relabeling of a single node is bounded by $O(n\epsilon)$, which immediately implies $O(nm)$ bound on the number of times a PUSH is executed during a single execution of GENERIC-REFINE *for the case of flows in networks with unit capacities.*

5.5.2 The Assignment Problem

The assignment problem is a weighted version of the bipartite matching problem. Similar to the unweighted case described in Section 5.4.2, we transform the assignment problem into a minimum-cost flow problem with unit capacities in the standard way (see, for example, [85]) where the weights on the edges are mapped into costs on the arcs of the corresponding matching network. As in Section 5.4.2, the nodes and edges in the resulting matching network are denoted by V and E, respectively. Without loss of generality, we assume that a perfect matching exists. To assure this we can always add a matching with arcs of very high cost.

The outer loop of the minimum-weight matching algorithm (see Figure 5.11) is the same as the outer loop of the Goldberg-Tarjan minimum-cost flow algorithm (see Figure 5.8.) The heart of the algorithm is the REFINE procedure, shown in Figure 5.12, that converts a 2ϵ-optimal flow into an ϵ-optimal flow. The procedure starts by decreasing the prices of all the nodes in T

```
procedure ASSIGNMENT($S, T, E, c$);

    [Initialization]
    transform the input problem into network flow form;

    $C \leftarrow \max\{|c(v,w)| : (v,w) \in E\}$;
    $\epsilon \leftarrow C$;
    for all $v \in V$ do $p(v) \leftarrow 0$;
    $p(s) = -2nC$;

    while $\epsilon \geq 1/n$ do
        $\epsilon \leftarrow \epsilon/2$;
        $(c, p, f) \leftarrow$ REFINE($V, E, c, p, f, \epsilon$);
    end;
    return the matching corresponding to current (maximum) flow $f$;
end.
```

Figure 5.11: High-level description of the outer (scaling) loop of the assignment algorithm.

by 2ϵ. (Though somewhat unnatural, this is essential for the proof of Lemma 5.5.4.) Next it constructs an ϵ-optimal pseudoflow by saturating residual arcs with reduced cost below $-\epsilon$ and creating appropriate excesses and deficits at the nodes.

The resulting pseudoflow is converted into an ϵ-optimal flow in two stages. The first stage iteratively uses the MATCH-AND-PUSH procedure (see Figure 5.5) to push the positive excesses towards the negative ones. The MATCH-AND-PUSH procedure used during this stage is exactly the same as for the unweighted case (see Figure 5.5), except that PUSH and RELABEL are generalized to the weighted case as described in Figure 5.10. We say that a node v is *active* if $e_f(v) > 0$ and the price change of this node during the current invocation of REFINE is below $k\epsilon$, where k is the *distance parameter*. (Note the similarity between this definition of the distance parameter and the definition which we have used in the algorithms for the unweighted case.) The first stage is executed as long as the number of active nodes exceeds l. (The best running time is achieved for activity parameter $l = \lfloor n^{2/3} \rfloor$ and distance parameter $k = \lfloor n^{1/3} \rfloor$.)

The second stage of REFINE finds augmenting paths from nodes with excess to the sink and augments along these paths. The paths used for the augmentations are shortest paths with respect to the distance function *length* obtained by adding ϵ to costs. Since we have assumed that the input graph has a perfect matching, REFINE terminates with a flow.

```
procedure REFINE($V, E, c, p, f, \epsilon$);
    [Reduce the number of arcs with negative reduced cost.]

    for all $v \in T$ do $p(v) \leftarrow p(v) - 2\epsilon$;

    [Convert into $\epsilon$-optimal pseudoflow.]

    for all $\{(v, w) \mid (v, w) \in E$ and $c_p(v, w) < -\epsilon\}$ do
        $f(v, w) \leftarrow f(v, w) + 1$;
        $e_f(w) \leftarrow e_f(w) + 1$;
        $e_f(v) \leftarrow e_f(v) - 1$;
    end;

    [First stage.]

    while the number of active nodes is at least $l$ do MATCH-AND-PUSH;

    [Second stage]

    for all $(v, w) \in E$ do $length(v, w) \leftarrow c(v, w) + \epsilon$;
    while there are active nodes do begin
        let $\Gamma$ be a shortest path w.r.t. $length$ from an active node to $t$;
        augment along $\Gamma$;
    end;

    return $(c, p, f)$;

end.
```

Figure 5.12: High-level description of the inner loop of the assignment algorithm. For the algorithm described in this chapter, we take $l = \lfloor n^{2/3} \rfloor$.

Since by construction the algorithm does not terminate as long as there are excesses, in order to prove the correctness of the algorithm it suffices to show that both stages preserve ϵ-optimality.

Lemma 5.5.2 Procedure MATCH-AND-PUSH preserves ϵ-optimality of the pseudoflow.

Proof: By construction the flow is pushed only through admissible arcs, and hence relabeling can not cause an increase in the price of a node. Assume that there exists a residual arc (v, w) after the execution of MATCH-AND-PUSH such that $c_p(v, w) < -\epsilon$. This means that when v was last relabeled either the price of w was lower, or this arc did not exist. The prices are nonincreasing, and hence the first case is impossible. If (v, w) was not in the residual graph when v was last relabeled, there was a PUSH from w to v since then. During this push the reduced cost of (v, w) was positive, which means that the price of w increased since then, which is impossible. ▮

The following lemma shows that the second stage preserves ϵ-optimality.

Lemma 5.5.3 Suppose a pseudoflow f is ϵ-optimal, and suppose $length : E \rightarrow R$ is defined by $length(v, w) = c(v, w) + \epsilon$. Let Γ be a shortest path with respect to *length* in G_f from a node with excess to t. Then augmentation along Γ preserves ϵ-optimality.

Proof: Relabel the graph by setting the price of each node to be equal to the negative of the distance from this node to the sink in the residual graph, where the distance is computed with respect to *length*. Observe, that the reduced costs of the residual arcs in this graph are at least $-\epsilon$. Moreover, for each node such that the sink is reachable from this node, there is a path to the sink along arcs with reduced cost of $-\epsilon$. ▮

The following lemma is needed to bound the residual flow after the first stage. This lemma is similar to the lemma due to Gabow and Tarjan [45], which they used to construct an $O(m\sqrt{n}\log nC)$-time sequential algorithm for weighted bipartite matching in the Hungarian framework. Our proof uses similar ideas but is more involved since in the push/relabel network we do not have a matching in the middle of the execution of the algorithm. Observe, that a direct consequence of our lemma is an $O(m\sqrt{n}\log nC)$-time push/relabel sequential algorithm for the minimum-weight bipartite matching problem.

Lemma 5.5.4 After the first stage of REFINE, the value of the residual flow is at most $O(n/k + l)$.

Proof: Consider nodes other then s and t that have excesses at the end of the first stage. We shall show below that the sum of price decreases at these nodes during the first stage is bounded by $O(n\epsilon)$. At the end of the first stage there can be at most l nodes with excesses such that their price was decreased by less than $k\epsilon$ during the first stage. Thus at most $O(n/k + l)$ nodes can have excess at the end of the first stage. Each excess has a value of 1, and therefore the total amount of excess at the end of the stage is $O(n/k + l)$. This gives the desired bound on the value of the residual flow.

We next show a bound of $O(n\epsilon)$ on the sum of price decreases at the nodes that have excesses at the end of the first stage. Intuitively, the main idea is as follows. Consider an ϵ-optimal pseudoflow f'_ϵ at some point during the execution of Refine and the ϵ-optimal flow f_ϵ at the end of the execution of Refine. The difference between f'_ϵ and f_ϵ can be decomposed into node-disjoint paths and cycles. The cost of pseudoflow f'_ϵ is lower than the cost of flow f_ϵ, and the difference in cost is equal to the sum of all the costs of these paths and cycles. On the other hand, this difference is "small" since both f_ϵ and f'_ϵ are ϵ-optimal. This leads to a bound on the sum of the costs of the paths. Now we observe that the cost of a path depends on the amount of relabeling of its end-point during the execution of Refine, which leads to the proof of the claim of the lemma.

The first step decreases the prices of all the nodes in T by 2ϵ. This increases the reduced costs of arcs that go into nodes in T by 2ϵ. The input is 2ϵ-optimal, and therefore after this increase all residual arcs that go into nodes in T have positive reduced cost. Hence the number of remaining residual arcs with negative reduced cost is at most n. (Recall that $n = |S| + |T|$.) On the other hand, the reduced cost of arcs going out of nodes in T is decreased by 2ϵ, and therefore the flow is 4ϵ-optimal with respect to the new prices.

For the purpose of the proof, we assume that at this point all costs are replaced by the reduced costs, and all prices are set to zero. We call the resulting costs *transformed*. After this transformation, the residual network has the following properties:

1. The (transformed) costs of residual arcs are at least -4ϵ.

2. At most n residual arcs have negative (transformed) costs.

Therefore, for any pseudoflow f in the network we have

$$\text{cost}(f) \geq -4n\epsilon. \tag{5.1}$$

To bound the cost of any ϵ-optimal flow f_ϵ from above, consider the decomposition of this flow into paths from s to t and cycles. The network is a matching network and therefore these paths and cycles are node-disjoint. The prices of both s and t are zero because they are never relabeled, and therefore the cost of f_ϵ is equal to the sum of the reduced costs of the saturated arcs, independent of the prices of the nodes other than s and t. Let p be the prices (associated with flow f_ϵ) at the end of the execution of Refine. For any saturated arc (v, w) (*i.e.*, $f_\epsilon(v, w) = 1$), there is a residual arc $(w, v) \in E_{f_\epsilon}$ with reduced cost $c_p(w, v) = -c_p(v, w) \geq -\epsilon$, and therefore we have

$$\begin{aligned} \text{cost}(f_\epsilon) &= \sum_{(v,w):f_\epsilon(v,w)=1} c_p(v, w) \\ &\leq n\epsilon. \end{aligned} \tag{5.2}$$

Consider an ϵ-optimal pseudoflow f'_ϵ and the associated prices p' at some point of the execution of the first stage. From (5.1) and (5.2) we have

$$\text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon) \leq 5n\epsilon. \tag{5.3}$$

Next we obtain a lower bound on $\text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon)$. Define the set of arcs $A' = \{(i, j) | f_\epsilon(i, j) > f'_\epsilon(i, j)\}$. Arcs in A' can be partitioned into simple paths from nodes with excess to nodes with deficit or to the sink t, simple paths from s to t, and simple cycles. Let S_1 be the set of paths from nodes with excesses to nodes with deficits or to t, let S_2 be the set of paths from s to t, and let S_3 be the set of cycles. Note that $A' \subseteq E_{f'_\epsilon}$ where $E_{f'_\epsilon}$ is the set of residual arcs of the pseudoflow f'_ϵ, and therefore the reduced cost (with respect to p') of any arc in A' is at least $-\epsilon$. Let $|P|$ denote the length of a path (or cycle) P. Then the cost of a path P from node v with excess to node w with deficit or to t is

$$\begin{aligned} \text{cost}(P) &= \sum_{e \in P} c(e) \\ &= \sum_{e \in P} c_{p'}(e) + p'(w) - p'(v) \\ &\geq -p'(v) - |P|\epsilon. \end{aligned}$$

The last inequality holds because neither t nor nodes with deficit are relabeled during execution of Refine. If P is either a path in S_2 or a cycle in S_3, then $\text{cost}(P) \geq -|P|\epsilon$. We have

$$\begin{aligned} \text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon) &= \sum_{P \in S_1} \text{cost}(P) + \sum_{P \in S_2} \text{cost}(P) + \sum_{P \in S_3} \text{cost}(P) \\ &\geq -\sum_{e(v)>0} p'(v) - \left(\epsilon \sum_{P \in S_1 \bigcup S_2 \bigcup S_3} |P| \right). \end{aligned}$$

By the properties of matching networks, the sum of the lengths of the paths in $S_1 \bigcup S_2$ and the cycles in S_3 is at most n. Therefore we have

$$\mathrm{cost}(f_\epsilon) - \mathrm{cost}(f'_\epsilon) \geq - \sum_{e(v)>0} p'(v) - n\epsilon. \tag{5.4}$$

From (5.3) and (5.4) we obtain

$$- \sum_{e(v)>0} p'(v) \leq 6n\epsilon.$$

This gives the desired bound on the sum of price decreases at the nodes that have excesses at the end of the first stage. Hence, the number of nodes that were relabeled by $k\epsilon$ during the first stage is at most $6n/k$. ▮

Lemma 5.5.4 bounds the number of iterations of the second stage of REFINE. Next we bound the number of iterations of the first stage. Note that the MATCH-AND-PUSH procedure maintains the invariants defined in Lemma 5.4.1.

Lemma 5.5.5 *There are at most $\frac{n(k+1)}{l} + l$ calls to* MATCH-AND-PUSH *in the first stage of* REFINE *where k and l are the distance and activity parameters, respectively.*

Proof: The proof is similar to the proof of Lemma 5.4.3. The main idea is that as long as there are at least l active nodes, prices of at least l nodes decrease by at least ϵ each during one invocation of MATCH-AND-PUSH. The lemma follows because the total amount of relabeling (price change) during the first stage of REFINE is bounded by $nk\epsilon$.

Consider an active node $v \in S$ that was matched with $w \in T$ during the first step of MATCH-AND-PUSH. If v pushes to w and w pushes to some node $v' \neq v$, the only residual arc from w after this push is (w, v). But the arc (v, w) was admissible, and therefore in the beginning of the iteration $c_p(v,w) = p(v) - p(w) + c(v,w) < 0$. By definition of RELABEL (see Figure 5.10), w is relabeled by

$$\begin{aligned}
\Delta p(w) &= p(w) + c(w,v) - p(v) + \epsilon \\
&\geq -c_p(v,w) + \epsilon \\
&\geq \epsilon.
\end{aligned}$$

Similarly, if the push was back to v, it is easy to see that w is also relabeled by at least ϵ.

Consider a node $v \in S$ that is not matched during the first step. From the previous discussion it follows that each node $w \in T$, such that (v, w) is a residual arc, is relabeled by at least ϵ. Hence, at Step 6, v is also relabeled by at least ϵ. ▮

Given the similarity of the above lemmas with the corresponding lemmas for the unweighted case, it is not surprising that the running times are similar.

Theorem 5.5.6 **The assignment algorithm runs in $O(n^{2/3} \log^3 n \log(nC))$ time using $SSP(n, m)$ processors.**

Proof: The initialization of Refine can be done in $O(\log n)$ time with $n + m$ processors. By Lemma 5.5.5, the first stage makes $O(\frac{nk}{l} + l)$ calls to Match-and-Push. By Lemma 5.4.2, Match-and-Push runs in $O(\log^3 n)$ time on a CRCW PRAM with $n + m$ processors. Hence, by Lemma 5.5.4, each execution of Refine takes $O((\frac{nk}{l} + l) \log^3 n)$ time on a CRCW PRAM using $SSP(n, m)$ processors.

By Lemma 5.5.1, after $O(\log(nC))$ iterations of Refine the resulting flow is optimal. The claim follows by setting $k = \lfloor n^{1/3} \rfloor$, $l = \lfloor n^{2/3} \rfloor$. ▮

5.5.3 Zero-One Minimum-Cost Flows

The ideas of the previous section can be extended to get a parallel algorithm that finds a maximum flow of minimum cost in graphs with zero-one capacities. The idea is to change the Refine procedure (see Figure 5.12) to use Push-and-Relabel procedure (see Figure 5.7) instead of Match-and-Push in the "while loop". The resulting algorithm runs in $O(m^{2/3} \log^2 n \log(nC))$ time using $O(SSP(n, m)$ processors.

Note, that slightly worse running time can be obtained by reducing the minimum-cost flow problem to weighted bipartite matching (see, for example, the paper of Chandra, Stockmeyer, and Vishkin [25]), and applying the algorithm described in the previous section. The transformation is as follows. First, construct a line digraph $G' = (V', E')$ of G, where

$$\begin{aligned} V' &= E \\ E' &= \{((u,v),(v,w)) | (u,v),(v,w) \in E\} \\ weight(((u,v),(v,w)) &= c(u,v) + c(v,w) \end{aligned}$$

Call the nodes of G' that correspond to arcs of G leaving the source or entering the sink "sources" and "sinks", respectively. Observe that there is a one-to-one correspondence between a minimum-cost flow in G and a maximum set of node-disjoint paths of minimum weight from sources to sinks in G'. Split each node v in G', except the sources or sinks, into two nodes v_1 and v_2. Connect these nodes by a zero-weight edge, and connect v_2 to u_1 for each arc (v,u) in G'. Add a complete bipartite graph between sources and sinks with edges of high weight. Call the resulting bipartite graph G''. It is easy to see that the minimum-weight perfect matching in G'' corresponds to the maximum set of node-disjoint paths of minimum weight in G', and therefore solves the minimum-cost flow problem for G. The number of nodes in G'' is $O(m)$, which leads to $O(m^{2/3}\log^3 n\log(nC))$ running time.

Remark: We can extend our results to networks with arbitrary integral capacities represented in unary by replacing every arc (v,w) of capacity $u(v,w)$ and cost $c(v,w)$ by $u(v,w)$ arcs of capacity one and cost $c(v,w)$.

5.6 Conclusions

The problems discussed in this chapter are important tools for design of efficient algorithms both in the context of parallel and sequential computation. For example, a linear-time sequential depth-first search algorithm leads to linear-time algorithms for many other problems. NC algorithms for the above problems would result in NC algorithms for many other problems as well. Our results are a step towards the design of efficient parallel algorithms for these problems. We believe that the ideas of this chapter may lead to improved sequential algorithms as well.

Our weighted bipartite matching algorithm works in two stages, where the first stage uses only a linear number of processors, and the second stage uses as many processors as needed to compute shortest paths in polylogarithmic time. Using a linear-processor connectivity algorithm in the second stage and rebalancing the distance and the activity parameters, leads to a linear-processor algorithm that solves the weighted bipartite matching problem in $O(n^{4/3}\log^3 n\log nC)$ time. The same transformation can be applied to our algorithm for finding minimum-cost flow in networks with unit capacities, leading to a linear-processor algorithm with $O((mn)^{2/3}\log^2 n\log nC)$ running time. Note that this algorithm works in the same time in a distributed network.

Chapter 6

Combinatorial Algorithms for the Generalized Circulation Problem

6.1 Introduction

Consider the following problem in financial analysis. An investor wants to take advantage of the discrepancies in prices of securities at different stock exchanges and of the currency conversion rates. His objective is to maximize his profit by trading at different exchanges and by converting currencies. The generalized circulation problem, considered in this chapter, models the above situation, assuming that a bounded amount of money is available to the investor and that bounded amounts of securities can be traded without affecting the prices.

The *generalized circulation problem* is a generalization of the maximum flow problem. Each arc (v, w) in the network has a *gain* factor $\gamma(v, w)$ and a *capacity* $u(v, w)$. If x units of flow enter the arc at v, then $x \cdot \gamma(v, w)$ units of flow arrive at w. The graph has a special node, called the *source*. The objective is to find a flow that maximizes the excess at the source. (Alternative formulations of the problem are discussed in Section 6.2.4.) In the model of the above financial analysis problem, nodes correspond to different currencies and securities, and arcs correspond to possible transactions. Gain factors represent the prices or the exchange rates. For example, an arc with gain factor of 113 from a vertex representing IBM stocks to a vertex representing U.S. dollars models the possibility of selling the stocks at $113 per share.

[0] The chapter represents joint work with A. Goldberg and É. Tardos [55, 54].

Generalized circulation problem can be used to model many situations which are impossible to express using standard network flows (see *e.g.*, [85, 48]). The gain factors can be used to represent the fact that we loose some fraction of the commodity while transporting it (due to damage, evaporation, theft, etc.) or that the commodity that enters an arc is transformed into a different commodity before leaving it (to model manufacturing processes, currency exchange, etc.).

The generalized circulation problem is in many respects similar to the minimum-cost circulation problem, where each arc has a cost per unit of flow in addition to capacity. This relationship, first observed by Onaga [106], has been studied in detail by Truemper [126]. Both problems can be viewed as problems of transporting a commodity from a producer to a consumer. Intuitively, the only difference between the two problems lies in the method of payment for shipping costs: in the case of the minimum-cost circulations, these costs are paid with money; in the case of the generalized circulations – with the commodity itself. The similarity is not only intuitive; the linear programming dual of these problems, as well as optimality conditions, are very similar.

Since generalized circulation problem is a special case of a linear program (LP), it can be solved in polynomial time using general-purpose linear programming algorithms, such as the ellipsoid method [79] or Karmarkar's algorithm [76]. Standard network flow problems are special cases of LP as well, and can be solved in polynomial time using general LP algorithms. They can also be solved by *combinatorial algorithms*, *i.e.* algorithms that exploit the combinatorial structure of the underlying network as opposed to being based on analytic ideas like the interior point methods for linear programming. The combinatorial approach to solve standard network flow problems proved to be extremely valuable and lead to algorithms with running times which are far superior to the running time bounds of the algorithms based on the general linear programming techniques.

Surprisingly, no polynomial-time combinatorial algorithm for the generalized circulation problem were known. Nevertheless, previous work on combinatorial algorithms for the generalized flow problem produced useful insights into the structure of the problem, which lead to combinatorial algorithms that run in finite time. Special variants of the linear programming simplex method have been developed for the generalized flow problem (see *e.g.*, [37]). These variants give combinatorial methods for solving the generalized network flow problem. Algorithms that iteratively augment the current flow have also been studied (see *e.g.*, [73, 74, 85, 112, 126]). However, no polynomial-time bounds are known for these algorithms.[1]

This chapter develops the machinery which allows us to design polynomial-time combinatorial algorithms for the generalized circulation problem. We present two algorithms, one based on the repeated application of a minimum-cost flow subroutine, and another based on the idea of augmenting along a biggest improvement path [36] and the idea of canceling negative cycles [60, 81]. We assume that the capacities are integers represented in binary, each gain is given as a ratio of two integers, and denote the value of the biggest integer used to represent the gains and capacities by B. Under these assumptions, the first algorithm runs in $O(n^2m(m + n\log n)\log n\log B)$ time and the second in $O(n^2m^2\log n\log^2 B)$ time.

The fastest general-purpose linear-programming algorithm currently known [127], when applied to the generalized circulation problem, runs in $O(m^4\log n\log B)$ time. Using techniques of Kapoor and Vaidya [75], this algorithm can be modified to take advantage of the special structure of the problem; the resulting algorithm runs in $O(n^{2.5}m^{1.5}\log n\log B)$ time [Vaidya, personal communication]. The running time bounds of our algorithms are better than the bounds achieved by the general-purpose linear programming algorithms. Although our time bounds are slightly worse than the time bound of Vaidya's specialization of his linear programming algorithm mentioned above, our algorithms lose by only a logarithmic factor on sparse graphs. Unlike Vaidya's algorithm, our approach exploits the combinatorial structure of the problem. Therefore we feel that it will lead to bounds that are better than those arising from implementations of general-purpose linear programming methods.

A remaining open question is the existence of a strongly polynomial algorithm for the generalized circulation problem. This problem is one of the simplest LP problems for which no strongly polynomial algorithm is known. (See [100, 123, 122] for a discussion of the classes of linear programs known to have strongly polynomial algorithms.) We believe that a combina-

[1] Prlat and Elmaghraby [112] claim to have developed a strongly polynomial algorithm based on iterative augmentation of flow. However, the proof in the paper has a major gap, and the method used for the augmenting path selection does not seem to be sophisticated enough to yield a polynomial-time algorithm.

torial approach is more likely to lead to a strongly polynomial algorithm for the generalized circulation problem, and view our results as a step in this direction.

This chapter is organized as follows. Section 6.2 defines the generalized circulation problem and closely related concepts, and gives their important algorithmic and combinatorial properties. Section 6.4 introduces the vertex labels, which are used to convert a given instance of the problem into an equivalent instance with several useful combinatorial properties. Section 6.5 reviews two simple combinatorial algorithms for the generalized circulation problem. Section 6.6 describes our first polynomial time algorithm, which is based on a minimum-cost flow subroutine. In Section 6.7 we present our second algorithm, based on the idea of augmenting the flow along a "big improvement" path. The last section contains concluding remarks.

6.2 Definitions and Background

In this section we review the definition of the minimum cost flow problem, define the generalized circulation problem, and discuss the relationship between them. In particular, we define the notion of Generalized Augmenting Path (GAP) and review the generalization of the flow-decomposition theorem for the case of generalized circulations.

6.2.1 Minimum-Cost Circulation Problem

First we discuss the minimum-cost circulation problem. A *circulation network* is a directed graph $G = (V, E)$ with arc capacities given by a nonnegative capacity function $u : E \rightarrow \mathbf{R}_\infty$.[2] We assume that G has no multiple arcs, *i.e.*, $E \subset V \times V$. If there is an arc from a node v to a node w, this arc is unique by the assumption, and we denote it by (v, w). This assumption is for notational convenience only. We also assume, without loss of generality, that the input graph G is symmetric: $(v, w) \in E \Longleftrightarrow (w, v) \in E$.

In the context of the minimum-cost circulation problem we need the following definitions. A *pseudoflow* is a function $f : E \rightarrow \mathbf{R}$ that satisfies the following constraints:

$$f(v,w) \leq u(v,w) \quad \forall (v,w) \in E \quad \text{(capacity constraints)}, \tag{6.1}$$

[2] $\mathbf{R}_\infty = \mathbf{R} \bigcup \{\infty\}$.

$$f(v,w) = -f(w,v) \quad \forall (v,w) \in E \quad \text{(flow antisymmetry constraints).} \tag{6.2}$$

Remark: To gain intuition, it is often useful to think only about the nonnegative components of a pseudoflow (or a generalized pseudoflow, defined in the next section). The antisymmetry constraints reflect the fact that a flow of value x going from v to w can be thought of as a flow of value $(-x)$ from w to v. The negative flow values are introduced only for notational convenience. Note, for example, that one does not need to distinguish between lower and upper capacity bounds: the capacity of the arc (v,w) represents a lower bound on the flow value on the opposite arc.

Given a pseudoflow f, the *residual capacity* function $u_f : E \to \mathbf{R}$ is defined by

$$u_f(v,w) = u(v,w) - f(v,w).$$

The *residual graph* with respect to a pseudoflow f is given by $G_f = (V, E_f)$, where $E_f = \{(v,w) \in E | u_f(v,w) > 0\}$. An *excess* $Ex_f(v)$ at a node v is equal to

$$Ex_f(v) = - \sum_{(u,v) \in E} f(u,v). \tag{6.3}$$

We will say that a node v has *excess* if $Ex_f(v)$ is positive, and has *deficit* if it is negative. A *circulation* is a pseudoflow with zero excess at every node:

$$Ex_f(v) = 0 \quad \text{(flow conservation constraints).} \tag{6.4}$$

A *cost function* is a real-valued function on arcs $c : E \to \mathbf{R}$. Without loss of generality, we assume that costs are antisymmetric:

$$c(v,w) = -c(w,v) \quad \forall (v,w) \in E \quad \text{(cost antisymmetry constraints).} \tag{6.5}$$

A cost of a circulation f is given by

$$c(f) = \sum_{(v,w) \in E : f(v,w) \geq 0} f(v,w) c(v,w).$$

The *minimum-cost circulation* problem is to find a minimum-cost (*optimal*) circulation in an input network.

Next we state two criteria for optimality of a circulation. Define the cost of a cycle to be the sum of costs of arcs along the cycle.

Theorem 6.2.1 (Busacker and Saaty [24]) A circulation is optimal if and only if its residual graph contains no negative-cost cycles.

To state the second criterion, we need the notions of the price function and the reduced cost function. A *price function* is a labeling on nodes $p : V \rightarrow \mathbf{R}$. A reduced cost function with respect to a price function p is defined by

$$c_p(v, w) = c(v, w) + p(v) - p(w).$$

These notions, which originate in the theory of linear programming, are crucial for many minimum-cost flow algorithms. As linear programming dual variables, node prices have a natural economic interpretation: they can be interpreted as current market prices of the commodity. We can interpret reduced cost $c_p(v, w)$ as the cost of buying a unit of commodity at v, transporting it to w, and then selling it.

Theorem 6.2.2 (Ford and Fulkerson [41]) The cost of a circulation f is minimum if and only if there is a price function p such that, for each arc (v, w),

$$c_p(v, w) < 0 \Rightarrow f(v, w) = u(v, w) \quad \text{(complementary slackness constraints).} \qquad (6.6)$$

Another useful fact about circulations is the decomposition theorem, which states that a circulation can be viewed as a collection of flows along cycles.

Theorem 6.2.3 (Ford and Fulkerson [41]) For every circulation f, there exists a collection of $k \leq m$ simple cycles $C_1, \ldots, C_k$ in G, and k positive numbers, $\delta_1, \ldots, \delta_k$, such that an arc (v, w) appears on one of the cycles only if $f(v, w) > 0$, and for every $(v, w) \in E$

$$f(v, w) = \sum_{i:C_i \text{ contains } (v,w)} \delta_i - \sum_{i:C_i \text{ contains } (w,v)} \delta_i.$$

6.2.2 The Generalized Circulation Problem

This section reviews the definitions of generalized pseudoflows, generalized circulations, and the generalized circulation problem. We summarize some combinatorial properties of the generalized pseudoflows and generalized circulations, and describe the correspondence between these properties and the properties of regular circulations and pseudoflows discussed in the previous section. For more details about the relationship between the minimum-cost flow and the generalized circulation, see [106, 126].

In the *generalized circulation problem*, every arc has a *gain* factor associated with it, with the gains given by a gain function $\gamma : E \rightarrow \mathbf{R}^+$.[3] We assume (without loss of generality) that the gain function is antisymmetric:

$$\gamma(v,w) = 1/\gamma(w,v) \quad \forall (v,w) \in E \quad \text{(gain antisymmetry constraints).} \tag{6.7}$$

Remark: Recall that the gain factors in our financial analysis problem correspond to currency exchange rates. Though the exchange rates are usually not antisymmetric, it is easy to use multiple arcs to describe them by a network that satisfies gain antisymmetry constraints.

In the case of ordinary flows, if $f(v,w)$ units of flow are shipped from v to w, $f(v,w)$ units arrive at w. In the case of generalized flows, if $g(v,w)$ units of flow are shipped from v to w, $\gamma(v,w)g(v,w)$ units arrive at w. A *generalized pseudoflow* is a function $g : E \rightarrow \mathbf{R}$ that satisfies the capacity constraints (6.1) and the generalized antisymmetry constraints:

$$g(v,w) = -\gamma(w,v)g(w,v) \quad \forall (v,w) \in E \quad \text{(generalized antisymmetry constraints).} \tag{6.8}$$

If $\gamma(v,w) > 1$, then (v,w) is a *gain* arc; if $\gamma(v,w) < 1$, then (v,w) is a *loss* arc. A gain of a path (cycle) is a product of gains of arcs on the path (cycle). Given a generalized pseudoflow g, the definitions of residual capacities, residual graph, and excesses are the same as for the ordinary pseudoflows.

A *generalized circulation* is a generalized pseudoflow that satisfies conservation constraints (6.4) at all nodes except at the *source* node.

[3] $\mathbf{R}^+$ denotes the set of positive real numbers.

The input to a *generalized circulation problem* is a tuple $(G = (V, E), u, \gamma, s)$, where G is a directed graph, u is a capacity function, γ is a gain function, and $s \in V$ is the *source*. For simplicity we assume that the capacities are finite and nonnegative. As before, for notational convenience, we assume that G is symmetric, has no multiple arcs, and that the residual graph of the zero flow is strongly connected. A *value* of a generalized pseudoflow g is the excess $Ex_g(s)$. The output of a generalized circulation algorithm is a generalized circulation of the highest possible value (an *optimal* generalized circulation).

Unless mentioned otherwise, we assume that the capacities are given as integers, each gain is given as a ratio of two integers, and that all integers are represented in binary. We denote the biggest integer used to represent capacities and gains by B. In addition, we denote the maximum of a product of gain numerators on a simple path or cycle in the input graph by T, and the least common multiple of numerators and denominators of all gains in the problem by L. Clearly, $T \leq B^n$ and $L \leq B^{2m}$. However, T and L are often significantly smaller than implied by the above bounds. In particular, this is the case for some of the problems obtained via reductions.

A *flow-generating cycle* is a cycle whose gain is greater than 1, and a *flow-absorbing cycle* is a cycle whose gain is less than 1. Observe that if one unit of flow leaves a node v and travels along a flow-generating cycle, more than one unit of flow arrives at v. Thus we can *augment* the flow along this cycle, that is, we can increase the excess at any node of the cycle while preserving excesses at other nodes by increasing flow along the arcs in the cycle and correspondingly decreasing flow on the opposite arcs, to satisfy the generalized antisymmetry constraints.

Recall the financial analysis interpretation of the generalized circulation problem, discussed in the introduction. From the investor's point of view, a residual flow-generating cycle is an opportunity to make profit. However, it is possible to take advantage of this opportunity only if there is a way to transfer the profit to the investor's bank account (the source node). This motivates the following definition.

A *generalized augmenting path (GAP)* is a residual flow-generating cycle and a (possibly trivial) residual path from a node on the cycle to the source. Given a generalized circulation and a GAP in the residual graph, we can augment the flow along the GAP, increasing the value of the current circulation. The role of GAPs in the generalized circulation problem is similar to the role of negative-cost cycles in the minimum-cost circulation problem: both can be used to augment the flow and thus improve the value of the current solution.

In the case of the maximum-flow problem, the flow is optimal if and only if the corresponding residual graph contains no augmenting paths [41]. A similar result holds for generalized circulations.

Theorem 6.2.4 (Onaga [106]) **A generalized circulation is optimal if and only if its residual graph contains no GAPs.**

Using the linear programming dual of the problem, it is possible to specify an alternate criterion of optimality, similarly to the way it is done for the minimum-cost circulation problem. We refer to the dual variables as prices. As in the context of the minimum-cost circulation problem, a *price function* p is a labeling of nodes by real numbers. In addition, in the context of the generalized circulation problem, we require $p(s) = 1$. Node prices can also be interpreted as market prices of the commodity at nodes, which motivates the definition of the reduced cost function. If a unit of flow is purchased at v and shipped to w, then $\gamma(v, w)$ units arrive at w. Buying the unit at v costs $p(v)$, and selling $\gamma(v, w)$ units at w returns $p(w)\gamma(v, w)$. Thus the reduced cost of (v, w) is defined as

$$c_p(v, w) = p(v) - p(w)\gamma(v, w).$$

Linear programming duality theory provides the following optimality criterion, which is similar to the one given by Theorem 6.2.2 for minimum-cost circulations.

Theorem 6.2.5 **A generalized circulation is optimal if and only if there exists a price function p, such that the complementary slackness conditions (6.6) hold for each arc $(v, w) \in E$.**

6.2.3 Decomposition of Generalized Pseudoflows

Theorem 6.2.3 states that a circulation can be decomposed into cycles. In this section we state a generalization of this theorem to generalized pseudoflows. We show how to decompose a generalized pseudoflow into "elementary" generalized pseudoflows.

We start by defining five types of elementary pseudoflows. Given a generalized pseudoflow g, let $\mathcal{E}(g)$ denote the set of nodes with excess and let $\mathcal{D}(g)$ denote the set of nodes with deficits. An elementary flow can be of one of the following types, where the type is defined according to the graph induced by the set of arcs on which this elementary flow is positive.

Type I A path from $\mathcal{E}(g)$ to $\mathcal{D}(g)$. It creates a deficit and an excess at the two ends of the path.

Type II A flow-generating cycle and a path connecting this cycle to a node in $\mathcal{E}(g)$. It creates excess at the end of the path. (If the path ends at the source, then this corresponds to a GAP.)

Type III A flow-absorbing cycle and a path connecting this cycle to a node in $\mathcal{D}(g)$. It creates deficit at the end of the path.

Type IV A cycle with unit gain. It does not create any excesses or deficits.

Type V A pair of cycles connected by a path, where one of the cycles generates flow and the other one absorbs it. It does not create any excesses or deficits.

Theorem 6.2.6 (Gondran and Minoux [66]) **A generalized pseudoflow g can be decomposed into components $g_1, \ldots, g_k$ with $k \leq m$ such that**

$$g(v,w) = \sum_i g_i,$$

where each component g_i is an elementary flow that is positive only on arcs (v,w) with $g(v,w) > 0$, and that belongs to one of the above mentioned types. This decomposition can be found in $O(nm)$ time.

Proof: We prove the theorem by induction on the number of arcs with nonzero flow value. Let G' denote the subgraph of G consisting of the arcs with positive flow value. If G' is acyclic, then a decomposition consisting only of paths (generalized pseudoflows of Type I) can be found by tracing the flow from some node with deficit to some node with excess. Otherwise, let C be a cycle in G'. If this cycle has gain 1, then we can subtract flow around the cycle until one arc on the cycle has zero flow value. The subtracted flow is of Type IV, and the theorem follows by induction.

Now consider the case when the gain of C is more than 1 (the case when the gain is less than 1 can be treated similarly). Decrease the flow along this cycle until one of the arcs along the cycle has zero flow value, creating deficit at one of the nodes v on the cycle, and denote the removed flow by h. Decompose $g-h$ according to the induction hypothesis. This decomposition includes several components that create the deficit at v. These components can be either of Type I or of Type III, and each is responsible for some amount of deficit, such that all the amounts sum up to the deficit at v in $g - h$. Observe that each one of these components, together with an appropriate fraction of h, corresponds to an element of Type II or Type V in the decomposition of g.

To establish the running time, observe that the above procedure decreases the number of arcs with positive flow value in amortized $O(n)$ time. ▮

6.2.4 Alternative Formulations

The generalized circulation problem can be stated in several ways. In this section we present different formulations of the problem and discuss the relationship among them. Understanding this relationship is important since different practical problems can be modeled in different terms, and certain algorithms seem more natural when applied to certain formulations.

Generalized Flow Problem (See [85] under the name of "flows with losses and gains".) The input to the problem is a graph $G = (V, E)$, a gain function $\gamma : E \rightarrow \mathbf{R}$, a source s, and a sink t. A *generalized flow* is a function on arcs that satisfies the antisymmetry constraints on all arcs and the conservation constraints on all nodes except s and t. The value of the flow is defined to be the amount of flow into the sink. Among all the generalized pseudoflows of maximum value, the goal is to find a generalized pseudoflow that minimizes the flow out of the source. The generalized flow problem is reducible to the generalized flows with profit.

Generalized Flows with Profit Problem (GFP Problem) The input to this problem is the same as the input to the generalized flow problem plus a number $r \in \mathbf{R}^+$ that gives the ratio of the price per unit of commodity at the sink to the price per unit of commodity at the source. The goal is to find a generalized flow that maximizes the profit. For a generalized flow g, the profit is $rEx_g(t) + Ex_g(s)$. To reduce a generalized flow problem to the GFP problem, chose r to be large enough (but finite); for example, $r = B^n + 1$.

The following linear-time reductions show that the GFP problem is equivalent to the generalized circulation problem. Given an instance $(G = (V, E), \gamma, s, t, r)$ of the GFP problem, we define an equivalent instance of the generalized circulation problem by adding an arc (t, s) with very large ($\geq nB^2$) capacity and a gain of r; and defining s to be the source. Given an instance $(G = (V, E), \gamma, s)$ of the generalized circulation problem, define an equivalent instance of the GFP problem by adding a new node t to the graph, along with the arcs (s, t) and (t, s), assigning unit gain and very high ($\geq nB^2$) capacity to these arcs, and letting $r = 1$.

6.3 The Restricted Problem

Instead of solving the generalized circulation problem directly, we will solve a restricted version of the problem. In this section we define the restricted problem and present $O(nm)$-time reduction of the general problem to the restricted one. Although similar restricted problems were considered before (see, for example, [73, 126]), we were unable to find a similar reduction in the literature.

We say that a generalized circulation network is *restricted* if in the residual graph of the zero flow all flow-generating cycles pass through the source. The *restricted problem* is a generalized circulation problem on a restricted network. The restricted problem has a simpler combinatorial structure and leads to simpler algorithms. Unless stated otherwise, in the subsequent sections we will refer to the restricted version of the problem under the name of the generalized circulation problem.

One of the nice facts about the restricted problem is that the optimality condition given by Theorem 6.2.4 simplifies in this case, and becomes very similar to Theorem 6.2.1. Before stating the simplified optimality condition, we prove a lemma that will also be useful later on.

Lemma 6.3.1 Let g be a generalized pseudoflow in a restricted network.

1. If the excess at every node other than at the source s is nonnegative, then for every node v there exists a (v, s)-path in the residual graph G_g.

2. If the residual graph of a generalized pseudoflow g has no flow-generating cycles and the excess at every node other than at the source s is non-positive, then for every node v there exists an (s, v)-path in the residual graph G_g.

Proof: The proof is by induction on the number of arcs with positive flow value. Recall that the residual graph of zero flow is strongly connected. Let G' be the subgraph induced by the arcs with positive flow value. Both statements are easy to prove when G' is acyclic. Now suppose G' contains a cycle C. This cycle is in the residual graph of the zero flow and therefore if its gain is more than 1, it has to pass through the source. Consider the case when the gain of C is at most 1. Decreasing the flow along C can only decrease the deficit at some node. The first claim follows by induction. Observe that if there are no flow-generating cycles in the residual graph of g, then the gain of C has to be at least 1, which proves the second claim. The case of C having a gain of more than 1 can be checked in a similar way. ▮

Theorem 6.3.2 (Onaga [105]) Given a restricted problem, generalized circulation g is optimal if and only if the residual graph of g contains no flow-generating cycles.

Proof: Clearly, if the residual graph of the generalized pseudoflow has no flow generating cycles, then it has no GAPs, and therefore it is optimal. To see the converse, consider a flow-generating cycle in the residual graph. We have to show that there is a path in the residual graph connecting this cycle to the source. The generalized circulation g has nonnegative excess at every node, and hence, by the the above lemma, there is a path from every node to the source. ▮

Corollary 6.3.3 Given a restricted problem, a generalized circulation g is optimal if and only if the residual graph of g has no negative-cost cycles, with $c = -\log\gamma$ as the cost function.

This corollary implies that the optimality of a solution for an instance of the restricted problem can be tested in one shortest-path computation.

Finally, we show how to reduce the generalized circulation problem to the restricted version of this problem.

Theorem 6.3.4 The generalized circulation problem is $O(nm)$-time reducible to the restricted problem.

Proof: The reduction works as follows. First, we saturate all gain arcs. More formally, define a generalized pseudoflow h by

$$\begin{aligned} h(v,w) &= u(v,w) && \text{if } (v,w) \text{ is a gain arc,} \\ h(v,w) &= -\gamma(w,v)u(w,v) && \text{if } (v,w) \text{ is a loss arc,} \\ h(v,w) &= 0 && \text{otherwise.} \end{aligned}$$

For every $v \in V$ such that $Ex_h(v) < 0$, add an arc (v,s) of gain $\gamma(v,s) = B^n + 1$ and capacity $u(v,s) = -Ex_h(v)$. (We also add reverse arcs with capacity 0 to preserve symmetry.) For every $v \in V$ such that $Ex_h(v) > 0$, add an arc (s,v) of gain $\gamma(s,v) = B^n + 1$ and capacity $u(s,v) = Ex_h(v)/\gamma(s,v)$ (and the reverse arcs with capacity 0 to preserve symmetry). Finally, define h to be zero on the new arcs. Let $\hat{G}$ be the extended graph. Then the transformed problem is $(\hat{G}, u_h, \gamma, s)$. Intuitively, the new arcs assure that nodes that have positive excess with respect to h are supplied with an adequate amount of "almost free" commodity, and nodes that have deficits with respect to h send adequate amount of commodity to s, whenever possible.

Define a pseudoflow g' by $g'(v,w) = \hat{g}(v,w) + h(v,w)$ for every $(v,w) \in E$. Observe that the residual graph of g' is the restriction of the residual graph of $\hat{g}$ to arcs in E. A node $v \in V - s$ has deficit in g' if and only if the arc (v,s) with gain $B^n + 1$ is in the residual graph of $\hat{g}$. Similarly, the node $v \in V - s$ has excess in g' if and only if the arc (s,v) with gain $B^n + 1$ is in the residual graph of $\hat{g}$. By Theorem 6.3.2, there are no flow-generating cycles in the residual graph of $\hat{g}$, which implies that there are no paths from nodes with excess to nodes with deficit in the residual graph of g'. Let S be the subset of nodes from which s is reachable in the residual graph of g'. Note, that there are no nodes with excess in $S - s$.

Decompose g' as described in Theorem 6.2.6. Since there is no path in the residual graph from nodes with excess to nodes with deficit, the decomposition does not include elements of Type I (paths). Therefore, excesses are created by flow-generating cycles, and deficits are created by flow-absorbing cycles. The existence of a flow-absorbing cycle in the decomposition implies that there are flow-generating cycles in the residual graph of g', which, by Theorem 6.3.2, contradicts the optimality of $\hat{g}$. This implies that g' cannot have deficits.

Subtract from the pseudoflow g' the elements of the decomposition that create the excesses at nodes other than s. Let g be the resulting generalized circulation. We claim that g is optimal. Observe that on arcs (v, w) inside S or entering S, the flow value $g(v, w) = g'(v, w) = \hat{g}(v, w) + h(v, w)$, and therefore there are no flow-generating cycles contained in S. Furthermore, the source s is not reachable from nodes outside S. Therefore the residual graph of g cannot contain a GAP. ▌

Note that if B is the biggest integer in the input problem, then the biggest integer in the transformed problem can be as high as $B^n + 1$. On the other hand, the transformation can not cause a significant increase in T and L: the corresponding parameters T' and L' of the transformed problem are bounded by $B^n(B^n + 1) \leq B^{2n+1}$ and $B^{2m}(B^n + 1) \leq B^{3m+1}$, respectively.

6.4 Vertex Labels and Equivalent Problems

In this section we review the idea of *relabeling*, which was originally introduced by Glover and Klingman [49] (under the name of scaling). We introduce the notion of *Canonical Relabeling*, and give an intuitive explanation of the relationship between the original and relabeled networks. We also prove an "integrality" theorem which will be used to prove termination of our polynomial-time algorithms, presented in subsequent sections.

Recall the financial analysis interpretation of the generalized circulation problem, described in the introduction, where vertices correspond to different securities or currencies, and arcs correspond to possible transactions. Suppose one country decides to change the unit of currency (for example, Great Britain could decide to introduce the penny as the basic currency unit, instead of the $\mathcal{L}$, or Italy could erase a couple 0's at the end of its bills). This causes an appropriate update of the exchange rates. Some of the capacities are changed as well (for example, a million $\mathcal{L}$ limit on the exchange from $\mathcal{L}$ to DM would now read as a limit of 100 million pennies).

The operation of changing the units of measure is called *relabeling* and the equivalent problem obtained by relabeling is called the *relabeled* problem. Let $\mu : V \longmapsto \mathbf{R}^+$ be a function denoting the number of old units per each new unit at nodes of the network. Given a function μ, we shall refer to $\mu(v)$ as the *label* of v.

Definition: Given a function $\mu : V \longmapsto \mathbf{R}^+$ and a network $N = (V, E, \gamma, u)$, the *relabeled network* is $N_\mu = (V, E, \gamma_\mu, u_\mu)$, where the *relabeled capacities* and the *relabeled gains* are defined by

$$\begin{aligned} u_\mu(v,w) &= u(v,w)/\mu(v) \\ \gamma_\mu(v,w) &= \gamma(v,w)\mu(v)/\mu(w). \end{aligned}$$

Given a generalized pseudoflow g and a labeling μ, the *relabeled residual capacity* is defined by:

$$u_{g,\mu}(v,w) = \frac{u(v,w) - g(v,w)}{\mu(v)}.$$

The following lemma relates pseudoflows in the original and the relabeled networks.

Lemma 6.4.1 If N is a generalized network and g is a generalized pseudoflow in N, then $g_\mu(v,w) = g(v,w)/\mu(v)$ is a generalized pseudoflow in the relabeled network N_μ. Moreover, the residual graphs of g and g_μ are the same.

6.4.1 Canonical Relabeling

Let g be a generalized pseudoflow whose residual graph has the property that all flow-generating cycles go through s (for example, the zero flow in the restricted problem). We shall use two symmetric ways to relabel a residual network. One is the *canonical relabeling from the source*, and the other is the *canonical relabeling to the source*. We use the first relabeling when we want to push additional flow from s, and the second when we want to push additional flow into s.

The canonical relabeling from the source applies when every node $v \in V$ is reachable from the source s via a path in the residual graph of the generalized pseudoflow g. For every node $v \in V$, the canonical label $\mu(v)$ is defined to be equal to the gain of the highest-gain simple s-v path in the residual graph. That is, one new unit corresponds to the amount of flow that can reach the node v if one old unit of flow is pushed along the most efficient simple path in the residual graph from s to v, ignoring capacity restrictions along the path.

Observe that the highest-gain path is the shortest path when arc lengths are defined to be $c(v,w) = -\log(\gamma(v,w))$. Because of the assumption that all flow-generating cycles pass through the source, the highest-gain path can be easily found by a single shortest-path computation, since deleting the arcs entering s from the residual graph yields a graph with no negative cycles.

The canonical relabeling to the source is defined similarly. It applies if there is a path in the residual graph from every node to the source s. The canonical label μ is defined as the inverse of the gain of the highest-gain simple v-s path in the residual graph.

The following important properties of the canonically relabeled problem are easy to prove. (Notice the symmetry between the theorems.)

Theorem 6.4.2 After a canonical relabeling from the source:

1. Every arc e with nonzero residual capacity, other than the arcs entering the node s, has $\gamma_\mu(e) \leq 1$.

2. For every node v, there exists a path from s to v in the residual graph with $\gamma_\mu(e) = 1$ for all arcs e on the path.

3. The most efficient flow-generating cycles consist of an (s,v)-path for some $v \in V$ with $\gamma_\mu(e) = 1$ along the path, and the arc $(v,s) \in E_g$ such that $\gamma_\mu(v,s) = \max(\gamma_\mu(e) : e \in E_g)$.

Theorem 6.4.3 After a canonical relabeling to the source:

1. Every arc e with nonzero residual capacity, other than the arcs leaving the node s, has $\gamma_\mu(e) \leq 1$.

2. For every node v, there exists a path from v to s in the residual graph with $\gamma_\mu(e) = 1$ for all arcs e on the path.

3. The most efficient flow-generating cycles consist of an (v, s)-path for some $v \in V$ with $\gamma_\mu(e) = 1$ along the path, and the arc $(s, v) \in E_g$ such that $\gamma_\mu(s, v) = \max(\gamma_\mu(e) : e \in E_g)$.

There is a simple correspondence between the units used for a relabeling and the prices from the linear programming dual of the problem. Intuitively, making the *unit at a vertex v smaller*, while keeping the price per unit constant, corresponds to increasing the price at v. In other words, changing the unit at v to $\mu(v)$ and keeping the price per unit ($p(v)$) constant has the same effect as keeping the size of the unit and setting the price to $p(v) = p(v)/\mu(v)$. If the price at every node is 1, then canonical relabeling from the source corresponds to changing the prices to $p(v) = 1/\mu(v)$. Note that, ignoring the arcs entering s, these prices are the marginal costs of the commodity, that is, the minimum prices (per unit) for which one could get some additional amount of the commodity to the nodes.

We can reformulate the optimality conditions of Theorem 6.2.5 to use labels instead of prices.

Lemma 6.4.4 A generalized circulation g in a restricted problem is optimal if and only if there exists a labeling μ such that every arc in the residual graph of the generalized circulation has $\gamma_\mu(e) \leq 1$.

We say that a labeling μ is *optimal* if there exist a generalized circulation g such that g and μ satisfy the conditions of Lemma 6.4.4.

Lemma 6.4.5 The optimality of a labeling μ for a restricted problem can be checked in one maximum-flow computation.

Proof: Relabel the network with labels μ. Suppose g is a generalized circulation that satisfies the optimality conditions with μ, and let g_μ be the corresponding generalized circulation in the relabeled network. If $\gamma_\mu(v,w) > 1$ then, by the optimality conditions, $g(v,w) = u(v,w)$, that is, $g_\mu(v,w) = u_\mu(v,w)$. Due to the symmetry, this uniquely defines the flow value on every arc with $\gamma_\mu(e) \neq 1$. The labeling μ is optimal if and only if g_μ can be extended to a feasible generalized circulation in the relabeled network. Hence, it is sufficient to solve a network flow feasibility problem on the subgraph of the relabeled network induced by the arcs with unit relabeled gain, which can be done via a single maximum flow computation. ∎

During a computation one has to keep the size of the numbers under control. The following lemma provides bounds on some of the numbers occurring in the algorithms.

Lemma 6.4.6 Let g be a generalized pseudoflow such that the canonical relabeling from the source (or to the source) in the residual graph applies. Let μ denote the canonical labels. Then $T^{-1} \leq \mu(v) \leq T$ and both numerator and denominator of $\mu(v)$ are divisors of L, for every $v \in V$.

Proof: The label $\mu(v)$ is equal to the gain of an s-v path in G, and hence satisfies the above claim, by the definitions of T and L. ∎

The following theorem will be used to prove termination for both of our polynomial-time algorithms. For a node v, a generalized pseudoflow g, and labels μ, let $Ex_{g,\mu}(v)$ denote the *relabeled excess* of the node v, that is, the total excess of the pseudoflow corresponding to g in the relabeled network with labels μ ($Ex_{g,\mu}(v) = Ex_g(v)/\mu(v)$). Consider the subgraph G' of G induced by the arcs with unit relabeled gain. By Lemma 6.4.6, in a canonically relabeled network the relabeled capacity of every arc with relabeled gain different from 1 is a multiple of L^{-1}. The next theorem states that the same is almost true for the arcs in G'. For any subset S, the relabeled capacity of arcs of G' entering S plus the relabeled excesses of the nodes in S is an integer multiple of L^{-1}.

Theorem 6.4.7 Let g be a generalized pseudoflow whose residual graph contains no flow-generating cycles. Let μ denote the canonical labels when relabeling from the source (or to the source) in the

residual graph of a generalized pseudoflow g. For any subset $S \subseteq V$,

$$X = \sum_{v \in S} Ex_{g,\mu}(v) + \sum_{v \in S; w \notin S \text{ and } \gamma_\mu(w,v)=1} u_{g,\mu}(w,v)$$

is an integer multiple of L^{-1}.

Proof: By Lemma 6.4.6, the labels, relabeled gain factors, and relabeled capacities, are multiples of L^{-1}. However, this is not necessarily true for the relabeled residual capacities. First consider an arc with relabeled gain higher than 1. By assumption, the flow on this arc is equal to the capacity, so the flow is a multiple of L^{-1}. By symmetry, the flow value on an arc (v, w), with relabeled gain of less than 1 is equal to $(-\gamma_\mu(w, v)u_\mu(w, v))$, so it is also a multiple of L^{-1}. This might not be true for arcs with unit relabeled gain.

The main observation is that the value of X is unchanged when the value of the flow is changed on an arc (and its opposite, to preserve the antisymmetry) with a unit relabeled gain. Indeed, if the arc e is contained in S, then changing the flow on e changes the excess at the two ends of the arc by opposite amounts. On the other hand, if e is outside of S, then it does not affect the expression.

Consider an arc e entering S. A change in the flow on e results in the change in the excess at the head of e and in the corresponding change in the residual capacity of e (in the opposite way). Consequently, one can replace the flow value on all arcs with unit relabeled gain by zero without changing the value of the expression. In the resulting generalized pseudoflow all flow values are multiples of L^{-1}, and therefore every term in the expression is an integer multiple of L^{-1}. ▌

6.5 Simple Algorithms

In this section we review two simple algorithms for the generalized circulation problem, where the first one due to Onaga [105] and the second one is due to Truemper [126]. Both algorithms are based on the natural approach of augmenting along flow-generating cycles. Though the algorithms are not efficient, we describe them in order to give some intuition for the polynomial-time algorithms presented in the subsequent sections and to acquaint the reader with our notation.

The first algorithm, due to Onaga [105], is similar to the minimum-cost flow method of Busacker and Goven [23] and Jewell [74], that augments the flow along a cheapest augmenting path. In the input network, all flow-generating cycles pass through the source. Onaga observed that this property is retained if the augmentation is done along the highest-gain flow-generating cycle. Onaga's algorithm iteratively augments the generalized circulation along highest-gain flow-generating cycles in the residual graph, until there are no such cycles left. By Theorem 6.4.2, if all flow-generating cycles pass through the source, then the highest-gain flow-generating cycle can be found by a shortest path computation. Therefore, each iteration of this algorithm consists of a shortest-path computation followed by an augmentation along a flow-generating cycle through the source.

By Theorem 6.3.2, we know that when the algorithm terminates, the resulting generalized circulation is optimal. Like the minimum-cost flow algorithm that augments the flow along the cheapest path, however, this algorithm does not run in finite time [41]. In order to make the running time finite, Truemper [126] uses a maximum-flow algorithm as a subroutine to augment flow along all of the highest-gain flow-generating cycles at once, instead of augmenting the flow on a cycle-by-cycle basis.

More precisely, consider the residual graph after the canonical relabeling from the source. Let $\alpha = \max\{\gamma_\mu(v,s) : (v,s) \in E_g\}$, and let N_α be the generalized network with underlying graph G_α that is induced by the arcs with unit relabeled gains and arcs $\{(v,s) \in E_g : \gamma_\mu(v,s) = \alpha\}$ together with their opposites. Then, by Theorem 6.4.2, all flow-generating cycles with the highest gain lie in G_α. Observe that any *nongeneralized* augmentation of flow in G_α, *i.e.*, augmentation that disregards the gain factors and views G_α as a standard network, corresponds to a valid *generalized* augmentation in G.

Lemma 6.5.1 A (ordinary) flow in G_α that maximizes the sum of the flow values on the arcs entering s with the gain factor α corresponds to an optimal generalized circulation in G_α.

Therefore, by a single maximum-flow computation, we can augment the flow so that there are no flow-generating cycles that pass through the source in G_α, and therefore there are no

```
Step 1 Find $\mu$, canonical labeling from the source.
       If $\gamma_\mu(v, w) \leq 1$ on every arc of the residual graph, then halt (the current circulation
       is optimal).
       Otherwise let $\alpha = \max_{e \in E_g} \gamma_\mu(e)$.
Step 2 Let $G_\alpha = (V, E_\alpha)$ be the network induced by residual arcs with either unit reduced
       gain or reduced gain of $\alpha$, and their opposites.
       Find a circulation $f'$ in $G_\alpha$ using the residual relabeled capacities, that maximizes.
       the flow on the arcs with reduced gain $\alpha$ in to $s$
Step 3 Update the current solution by setting $g(v, w) = g(v, w) + f'(v, w)\mu(v)\ \forall(v, w) \in$
       $E; v \neq s$, and set the value $g(s, w)\ \forall w \in V$ to keep the symmetry.
```

Figure 6.1: A single iteration of the maximum-flow based algorithm.

flow-generating cycles with gain α in the residual graph.

The maximum-flow based algorithm proceeds in iterations, where at each iteration we compute the canonical relabeling from the source, construct G_α, compute the appropriate maximum-flow, and interpret the result as an augmentation of the current generalized circulation. See Figure 6.1 for a more formal description. All flow-generating cycles in the residual graphs of the generalized circulations found throughout the algorithm pass through the source, which guarantees that the canonical relabeling is applicable. The optimality of the solution produced when the algorithm terminates follows from Theorem 6.3.2.

Theorem 6.5.2 The highest gain of a flow-generating cycle in the residual graph is strictly decreasing from one iteration to another, and therefore the number of iterations is bounded by the number of different gains of simple flow-generating cycles in the original network.

Corollary 6.5.3 If the gain factors in the input are all integer powers of 2 (or of any other constant), then the above algorithm runs in polynomial time.

Observe, that the algorithm which is based on path by path augmentation may not terminate, whereas the maximum flow-based algorithm terminates in exponential time. In the next two sections we describe more efficient algorithms for the problem.

6.6 Algorithm MCF

In this section we present our first algorithm that solves the restricted version of the generalized circulation problem in polynomial time. Unless stated otherwise, we will refer to the restricted problem as the generalized circulation problem. This algorithm is based on a minimum-cost flow subroutine, and we call it Algorithm MCF. The main idea of the algorithm is best described by contrasting Algorithm MCF with the maximum-flow based algorithm, presented in the previous section. At each iteration, both algorithms solve a simpler flow problem, and interpret the result as an augmentation in the generalized circulation network. The maximum-flow based algorithm is slow because at each iteration it considers only arcs with unit relabeled gains and some of the arcs adjacent to the source, disregarding the rest of the graph completely. The algorithm presented in this section considers all arcs, assigns cost $c(e) = -\log \gamma_\mu$ to each arc, and solves the resulting minimum-cost circulation problem (disregarding flow gains and losses).

The *interpretation* of a pseudoflow f is a generalized pseudoflow g, such that

$$g(v,w) = \begin{cases} f(v,w) & \text{if } f(v,w) \geq 0, \\ -\gamma_\mu(w,v)f(w,v) & \text{otherwise} \end{cases}$$

Note that, as opposed to the case of the maximum-flow based algorithm, where the interpretation of a feasible circulation is a feasible generalized circulation, the interpretation of a minimum-cost circulation leads to a generalized pseudoflow. Whenever the circulation uses arcs with relabeled gain of less than 1, its interpretation has deficits. A connection between a pseudoflow f and its interpretation is given by the following lemma.

Lemma 6.6.1 The residual graphs of a pseudoflow f and its interpretation g as a generalized pseudoflow are the same.

The algorithm starts with the zero generalized pseudoflow, which has flow generating cycles in the residual graph. By the definition of the restricted problem, all these cycles path through the source. Using this fact, we will show that the only iteration that creates a positive excesses is the first one, and that the only positive excess created is the one at the source. Each subsequent iteration tries to use this excess to balance deficits at various nodes of the graph, created by

```
Step 1 Find $\mu$, canonical labeling from the source.
       If $\gamma_\mu(v, w) \leq 1$ on every arc of the residual graph and $\forall v \in (V - s) : Ex_{g,\mu}(v) = 0$,
       then halt (the current circulation is optimal).
Step 2 Introduce costs $c(v, w) = -\log \gamma_\mu(v, w)$ on the arcs of the network.
       Find a minimum cost pseudoflow $f'$ in the residual relabeled network $G$ that has
       excess $Ex_{f'}(v) = - Ex_{g,\mu}(v)$ for every node $v \in V - s$.
Step 3 Let $g'$ be the interpreted version of $f'$.
       Update the current solution by setting $g(v, w) = g(v, w) + g'(v, w)\mu(v)\ \forall (v, w) \in E$.
```

Figure 6.2: Inner loop of Algorithm MCF.

interpretation of flow through arcs with relabeled gain of less than 1. In each iteration we find a minimum-cost flow that satisfies the deficits that were left after the previous iteration.

Algorithm MCF, shown in Figure 6.2, maintains a generalized pseudoflow g in the original (nonrelabeled) network, such that the excess at every node other than the source is nonpositive. The algorithm proceeds in iterations. At each iteration it canonically relabels the residual graph, solves the corresponding minimum-cost flow problem in the relabeled network, and interprets the result as a generalized augmentation.

6.6.1 Analysis of the Inner Loop of the Algorithm

In this section we prove that each iteration of Algorithm MCF can be implemented to run in in polynomial time, and show that the algorithm produces an optimal generalized circulation upon termination. The proof that the number of iterations is polynomial is deferred until the next section.

The most important property of a minimum-cost flow, for this application, is that its residual graph has no negative cycles. By Lemma 6.6.1 this yields the following corollary:

Corollary 6.6.2 The residual graphs of the generalized pseudoflows introduced by the algorithm in Step 3 have no flow-generating cycles.

Lemma 6.6.3 **The following statements are true for a generalized pseudoflow g that is constructed by Step 4:**

1. **The canonical relabeling applies to the residual graph of g.**

2. **All excesses, except at the source, are nonpositive.**

Proof: We will prove the lemma by induction on the number of iterations. Assume that at the end of an iteration we have a generalized pseudoflow g that satisfies the statements of the lemma. We shall prove that the statements are satisfied at the end of the next iteration.

By Corollary 6.6.2, the residual graph of g has no flow-generating cycles, and therefore, by Lemma 6.3.1, every node is reachable from the source in the residual graph of g. Hence, we can compute the canonical relabeling from the source at Step 1, which proves the first statement of the lemma. Moreover, the absence of flow-generating cycles in the residual graph means that there are no arcs with relabeled gain of more than 1 in this graph. Therefore, the interpretation of flow f', computed at Step 2, can create only deficits, which proves the second statement.

To prove the correctness of the statements after the first iteration, recall the assumption that the residual graph of the zero generalized pseudoflow is strongly connected. Also, by definition of the restricted problem, all flow-generating cycles in the residual graph of the zero generalized pseudoflow pass through the source. Therefore we can apply canonical relabeling from the source. Moreover, after the relabeling, all arcs with relabeled gain of more than 1 enter the source. Therefore, the only node that can have a positive excess after the interpretation of the flow f' is the source. ▮

Lemma 6.6.4 **For a generalized pseudoflow g and the labeling μ after Step 1, there exists a pseudoflow f' in the relabeled residual network of g with $Ex_{f'}(v) = -Ex_g(v)$ for every node $v \in V - s$.**

Proof: Let g be a generalized pseudoflow at the beginning of an iteration. Decompose it according to Theorem 6.2.6. The only elements of the decomposition that can contribute to deficits are paths from nodes with deficits to the source and paths from nodes with deficits to flow-absorbing cycles. Existence of a flow-absorbing cycle in the decomposition implies that there are flow-generating cycles in the residual graph of g, which contradicts Corollary 6.6.2, and therefore there are no flow-absorbing cycles in the decomposition.

Consider a subset of the nodes S containing the source. By the previous lemma, the only node that can have a positive excess is the source, and therefore it is sufficient to prove that the relabeled residual capacity of the cut defined by S exceeds the sum of the relabeled deficits.

Consider an (s, v)-path in the decomposition where $v \notin S$, and let (w_1, w_2) be an arc on this path that leaves S. Recall that one of the properties of the decomposition is that (w_2, w_1) is an arc of the residual graph. By Corollary 6.6.2, there are no flow-generating cycles in the residual graph of g, and hence the relabeled gain of every arc in the residual graph of g is at most 1. Therefore, the sum of deficits created by the paths in the decomposition that use this arc, is at most the relabeled residual capacity of the opposite arc (w_2, w_1). ▮

The above lemmas show that the algorithm can proceed until a generalized circulation is found. By Corollary 6.6.2 and Theorem 6.3.2, the generalized circulation found is optimal. Hence, we have proved the following theorem.

Theorem 6.6.5 **Each iteration of the algorithm can be implemented in polynomial time, and the generalized circulation g, produced by the algorithm upon termination, is optimal.**

6.6.2 Bounding the Number of Iterations

Consider a generalized pseudoflow g at the beginning of an iteration. The fact that there are deficits and that there are no flow-generating cycles in the residual graph means that the current excess at the source is an overestimate on the value of the maximum possible excess. It is easy to see that the sum of the deficits (after the relabeling) at all the nodes except at the source is a lower bound on the amount of the overestimation. This suggests to use this value, $Def(g, \mu) = \sum_{v \neq s}(-Ex_{g,\mu}(v))$, as a measure of the proximity of a generalized pseudoflow to an optimal generalized circulation.

First we show that if $Def(g, \mu)$ is very small, then the algorithm terminates after one more iteration.

Theorem 6.6.6 If $Def(g,\mu) < L^{-1}$ before Step 2 of an iteration, then the algorithm produces an optimal generalized circulation at the end of Step 3 of this iteration.

Proof: We claim that the pseudoflow f', computed at Step 2 of this iteration, uses zero cost arcs only. To see this, consider a set of nodes S containing the source s. We have to argue that the sum of the relabeled deficits of the nodes not in S is at most equal to the sum of the relabeled residual capacities of the zero cost arcs entering S. Clearly, the difference is at most $Def(g,\mu) < L^{-1}$. Applying Theorem 6.4.7 to the complement of S shows that this difference is an integer multiple of L^{-1}. Therefore, it is nonpositive.

The interpretation g' of a pseudoflow f' that uses zero cost arcs only, is equal to the pseudoflow. Therefore, no deficits are left after Step 3. The lemma follows because the residual graph contains no flow-generating cycles throughout the algorithm (by Corollary 6.6.2). ▮

An important observation is that the labels μ are monotonically decreasing during the algorithm. The next lemma relates the decrease in the labels to the price function from the minimum-cost flow computation. Let p' denote the optimal price function associated with the pseudoflow f' found in Step 2. Assume, without loss of generality, that $p'(s) = 0$.

Lemma 6.6.7 Let f' be the minimum-cost pseudoflow found in Step 2 of an iteration, and let p' be the associated price function. For each node v, the canonical relabeling in Step 1 of the next iteration decreases the label $\mu(v)$ by at least a factor of $2^{p'(v)}$.

Proof: The decrease in the label of a node v is computed by finding a shortest path in the residual graph from s to v. The price function p' is optimal, and hence the reduced costs of the arcs in the residual graph are nonnegative. Therefore, the length of any (s,v) path is at least $p'(v) - p'(s)$. ▮

For the analysis of the algorithm we decompose the minimum cost flow into paths, and consider the paths one by one. The decomposition is based on the following lemma.

Lemma 6.6.8 (Ford and Fulkerson [41]) Let f be a minimum-cost flow that satisfies given (nonnegative) demands at every node other than s and let p be an optimal price function such that $p(s) = 0$. The flow f can be decomposed into cycles and paths from s to the other nodes, such

that the cycles and the paths are in the residual graph of the zero flow, the cycles have nonpositive cost, and the cost of the paths ending at a node v is at most $p(v)$.

Proof: For every node $v \in V$ add the arcs (v,s) and (s,v). Extend the flow f to these arcs, so that it becomes a circulation. Define the cost of an arc (v,s) to be equal to $c(v,s) = -p(v)$. The new arcs have zero reduced cost, and hence the circulation is of minimum cost.

The lemma is proved by applying Theorem 6.2.3 to decompose this circulation into cycles. Note that the arcs opposite to the ones that belong to the cycles of the theorem are in the residual graph of the circulation. Therefore, the cycles have nonpositive cost. Any cycle that uses a new arc (v,s) corresponds to (s,v)-paths in the original graph. ▮

The key idea of the analysis is to distinguish two cases: Case 1, where the flow f' can be decomposed into "cheap" paths, (*e.g.*, $p'(v)$ is small, say $p(v) < \log 1.5$, for every $v \in V$); and Case 2, where there exists $v \in V$ such that $p'(v)$ is "large" ($\geq \log 1.5$). We show that in the first case $Def(f,\mu)$ decreases significantly, while in the second case, by Lemma 6.6.7, some of the labels decrease significantly. Using Theorem 6.6.6 and Lemma 6.4.6, we prove that neither case can occur too many times.

The following lemma is used to estimate the total deficit created when interpreting a flow as a generalized pseudoflow.

Lemma 6.6.9 **Let f' be a flow along a simple path P from s to some other node v that satisfies one unit of deficit at v. Let g' be the interpretation of f' as a generalized pseudoflow. Assume that all relabeled gains along the path P are at most 1, and denote them by $\gamma_1,\ldots,\gamma_k$. Then after augmenting by g', the unit of deficit at v is replaced by deficits that sum up to at most $(\prod_{1\leq i\leq k}\gamma_i)^{-1} - 1$.**

Proof: The deficit created at the ith node of the path is $(1-\gamma_i)$, for $i = 1,\ldots,k$. Using the assumption that the gain factors along the path are at most 1, the sum of the deficits can be bounded by

$$\begin{aligned}\sum_{i=1}^{k}(1-\gamma_i) &\leq \sum_{i=1}^{k}\frac{1-\gamma_i}{\prod_{j=1}^{i}\gamma_j}\\ &= \frac{1}{\prod_{i=1}^{k}\gamma_i} - 1.\end{aligned}$$

▮

Using this lemma and Theorem 6.2.6, we bound the value of $Def(g,\mu)$ after an application of Step 3. Let p' be an optimal price function associated with the flow f', such that $p'(s) = 0$. Let $\beta = \max_{v \in V} p'(v)$ denote the maximum price.

Lemma 6.6.10 The value of $Def(g,\mu)$ after an application of Step 3 can be bounded by $2^{\beta} - 1$ times its value before the Step.

Proof: Use Lemma 6.6.8 to decompose f' into paths and cycles. Interpret the pseudoflow f' by interpreting the cycles and the paths one by one. The costs associated with the arcs of the residual graph of the generalized pseudoflow g are nonnegative, and therefore the cycles in the decomposition consist of zero-cost arcs only. Interpretation of the flow along these cycles does not change the flow value; in particular it does not create deficits. By Lemma 6.6.9, when interpreting a path from s to v, the deficit at v is replaced by deficits that sum up to at most $2^{p'(v)} - 1 \leq 2^{\beta} - 1$ times the deficit at v satisfied by this path. ▮

Corollary 6.6.11 If $p'(v) < \log 1.5$ for every node v, then $Def(g,\mu)$ decreases by a factor of 2 during the application of Step 3.

The remaining difficulty is the fact that the function $Def(g,\mu)$ can increase, either when the canonical relabeling is done in Step 1 or in Step 3 when Case 2 applies. If $Def(g,\mu)$ has increased by some factor during relabeling (Step 1), or after interpretation of f' (Step 3), however, then at least one of the nodes is relabeled by the same factor, as it is shown by the following lemma.

Lemma 6.6.12 If either during Step 3 of one iteration or Step 1 of the next one, $Def(g,\mu)$ increases by a factor of α, then the label of some node decreases by at least a factor of α during Step 1 of the next iteration.

Proof: The increase in $Def(g,\mu)$ during Step 1 is due to relabeling, and hence the deficit at a node can increase during this Step by a factor of α if and only if the label at this node decreases by the same factor. By Lemma 6.6.10, the increase in $Def(g,\mu)$ during Step 3 is bounded by 2^{β}, where $\beta = \max p'(v)$. Hence, if $Def(g,\mu)$ increases by α during this step, there exists a node v such that $p'(v) \geq \log \alpha$. By Lemma 6.6.7, this means that $\mu(v)$ decreases by at least α during Step 1 of the next iteration. ▮

Combining the above results, we can bound the overall growth of $Def(g, \mu)$ during an execution of the algorithm.

Lemma 6.6.13 The growth of the function $Def(g, \mu)$ throughout an execution of the algorithm is bounded by a factor of $T^{O(n)}$.

Proof: The function $Def(g, \mu)$ can increase either as a result of Step 1 or Step 3. By the above lemma, such increase is followed by a decrease in the label of one of the nodes by at least the same factor during the subsequent relabeling. The claim follows from Lemma 6.4.6, that limits the decrease of the label of any node. ▮

Due to Lemma 6.4.6, Case 2 cannot occur too many times. The above lemma helps to bound the number of times Case 1 can occur.

Theorem 6.6.14 The algorithm terminates in $O(n \log T) = O(n^2 \log B)$ iterations.

Proof: Lemma 6.4.6 shows that Case 2 cannot occur more than $O(n \log T)$ times. When Case 1 applies, the value of $Def(g, \mu)$ decreases by a factor of 2. The value of $Def(g, \mu)$ is at most $O(nBT)$ after the first iteration, and by Theorem 6.6.6, the algorithm terminates when this value decreases below L^{-1}. Lemma 6.6.13 limits its increase during the algorithm. Hence, Case 1 cannot occur more than $O(n \log T)$ times. ▮

To get a bound on the running time, we must decide which minimum-cost flow algorithm to use as a subroutine. The best choice turns out to be Orlin's strongly polynomial algorithm [107].

Theorem 6.6.15 The above generalized flow algorithm can be implemented so that it will use at most $O(n^2 m(m + n \log n) \log n \log B)$ arithmetic operations on numbers whose size is bounded by $O(m \log B)$.

Remark: The choice of a strongly polynomial minimum-cost flow algorithm is somewhat surprising, since our algorithm is not strongly polynomial. The reason for this choice is that the current best scaling algorithms would give worse running times, even when the size of the numbers in the input is small. Observe that the number of bits needed to represent the capacities of the intermediate minimum-cost flow problems can be as high as $m \log B$, which would make a capacity-scaling algorithm too slow. The classical cost-scaling algorithms cannot be used directly because the costs are (irrational) logarithms of rational numbers. A cost-scaling algorithm can be constructed based on the idea of ϵ-optimality, as done in [58, 64]. Note that the gain of a flow-generating cycle can be as small as $2^{-O(B^n)}$, and hence the absolute value of the cost of a negative cycle can be as small as $B^{-O(n)}$. Consequently, the required precision seems to be $\Omega(B^{-n})$, and therefore the cost-scaling algorithms are too slow as well.

6.7 The Fat-Paths Algorithm

Recall that a GAP is defined to be a flow-generating cycle with an augmenting path connecting this cycle to the source. A natural way to make progress towards an optimal solution is to increase the flow along a GAP. We call this a *generalized augmentation.* Clearly, if the augmentations are done along arbitrary GAPs, we do not get a polynomial-time algorithm. One way to improve the running time is to execute only those generalized augmentations that result in a significant progress towards an optimal solution.

We divide a generalized augmentation into two parts: augmenting the flow along a flow-generating cycle and bringing the created excess to the source. Our algorithm first saturates *all* flow-generating cycles, creating excesses at various nodes of the graph, and only then looks for augmenting paths from nodes with excess to the source. This method does not create deficits. A natural way to measure the progress of this algorithm is by the difference between the excess at the source in the current generalized pseudoflow and the excess at the source in the optimal solution. This difference is called the *excess discrepancy.* We shall show that if the excess discrepancy is very small, a single maximum-flow computation produces an optimal solution (similarly to Theorem 6.6.6).

Consider an augmenting path with an excess in the beginning of the path. Even if this excess is very large, the increase in the excess at the source caused by an augmentation along this path is limited, and depends on both the capacities and the gains of arcs on the path. We call this limit the *fatness* of the path. A simple v-s path is Δ-*fat* if, given unlimited supply at v, at least Δ units of flow can be sent to s along this path. An arc is Δ-fat if it belongs to a Δ-fat path. Saturating all the flow-generating cycles before bringing the excesses to the source facilitates the search for Δ-fat paths.

The section consists of three parts. In the first part we describe the *Fat-Path* algorithm, assuming existence of the *Fat-Augmentation* and the *Cancel-Cycles* procedures, and discuss the correctness and the running time of the algorithm. The second part presents the *Fat-Augmentation* procedure. This procedure finds a Δ-fat path in the residual graph and augments the flow along it. The third part of this section is devoted to the description of the *Cancel-Cycles* procedure, which transforms a generalized pseudoflow with flow-generating cycles in its residual graph into a generalized pseudoflow with no such cycles, without creating any deficits.

6.7.1 Fat-Path Algorithm - Overview

The Fat-Path algorithm, described in Figure 6.3, solves the restricted problem in phases. The goal of each phase is to decrease the bound on the excess discrepancy by at least a constant factor. The input and output of a phase is a generalized pseudoflow with non-negative excesses.

Each phase consists of three stages. The input to the first stage is a generalized pseudoflow with non-negative excesses and, possibly, with flow-generating cycles. This stage uses procedure *Cancel-Cycles*, described in Section 6.7.3, to cancel all such cycles, creating new excesses at various nodes of the graph.

In the second stage we test whether it is possible to bring all the excesses to the source over the most efficient residual paths. This test is performed by canonically relabeling to the source and then computing a maximum flow in the graph induced by the unit relabeled gain arcs. Observe that by Lemma 6.3.1, the canonical relabeling to the source applies. There are no flow-generating cycles in the residual graph after the first stage, and an augmentation of the flow along arcs with a unit relabeled gain does not create new ones. Therefore, if we succeed in bringing all excess to the source using these arcs, the resulting generalized circulation is optimal, and the algorithm terminates. This stage may be viewed as a variation of a single iteration of the maximum-flow based algorithm, described in Section 6.5.

```
Stage 1 Cancel all flow-generating cycles by calling the Cancel-Cycles procedure.
Stage 2 Compute $\mu$, the canonical labeling to the source. Let $E'$ be the set of all arcs with
        unit relabeled gain. Consider nodes with excess as sources with capacity bounded by
        the value of their relabeled excess, and compute maximum flow $f'$ from these nodes
        to the source in the graph induced by the arcs in $E'$ with the relabeled residual
        capacities. Update the generalized pseudoflow $g$ by setting $g(v,w) = g(v,w) +$
        $f'(v,w)\mu(v)\ \forall(v,w) \in E'$.
        Terminate if $\forall v \in V - s:\ Ex_g(v) = 0$.
Stage 3 Repetitively call Fat-Augmentation procedure to augment the flow along $\Delta$-fat paths
        from nodes with excess to the source, as long as such paths exist.
        Set $\Delta = \Delta/2$.
```

Figure 6.3: A single phase of the Fat-Path algorithm.

The input to the third stage is a generalized pseudoflow with nonnegative excesses and no flow-generating cycles, and a "fatness" parameter Δ (initially set to B^2). This stage iteratively augments the flow on Δ-fat paths from nodes with excess to the source, reducing the excess at the starting node of the path and increasing the excess at the source. The stage continues until there are no more Δ-fat paths from nodes with excess to the source. Before the next phase, the value of Δ is decreased by a factor of 2. We shall show that this stage does not introduce flow-generating cycles in the graph induced by the Δ-fat arcs.

The following lemma indicates that the excess discrepancy is a good measure of progress.

Lemma 6.7.1 If the excess discrepancy is below L^{-1}, and there are no negative excesses or flow-generating cycles, then all the excesses are brought to the source by Stage 2 of the Fat-Path algorithm.

Proof: We claim that after Stage 2 the excess discrepancy at the source is an integer multiple of L^{-1}. This means that the excess discrepancy decreases by at least L^{-1} in between any two iteration, which directly implies the lemma.

Consider the generalized pseudoflow g before Stage 2. Let $(S, \overline{S})$ be the cut saturated by the maximum flow computation in Stage 2, such that there are no excesses left in S and $s \in S$. Let μ denote the canonical labels computed during this stage. After this stage, the excess at the source is equal to the sum of $Ex_{g,\mu}(v)$ for $v \in S$ plus the sum of the relabeled residual capacities of the arcs with relabeled gain 1 entering S, *i.e.* all the excesses in S are brought to the source. Applying Theorem 6.4.7 to $V' = S$, the pseudoflow g, and the labels μ, we conclude that this sum is an integer multiple of L^{-1}. ▮

The following lemma bounds the excess discrepancy in terms of Δ.

Lemma 6.7.2 The excess discrepancy at the end of a phase is at most $O(m\Delta)$.

Proof: Let g be a generalized pseudoflow at the end of a phase and let g^* be an optimal generalized circulation. Let $E^+ = \{e|g^*(e) > g(e)\}$ be the arcs with positive residual flow, and let $G^+ = (V, E^+)$. The residual pseudoflow $g^* - g$ can be decomposed according to Theorem 6.2.6. The arcs opposite to the ones used by the cycles in the decomposition are in the residual graph of the generalized circulation g^*, and therefore the decomposition consists only of paths from nodes with excess to the source and GAPs. Note that each one of these paths and GAPs is in G^+. There are at most $O(m)$ paths and GAPs used, and therefore it is sufficient to show that each one of them contributes at most Δ to the excess at the source.

First, consider paths from excesses to the source. By construction, no path in G_g from excess to the source is Δ-fat. Also, we have $E^+ \subseteq E_g$. Hence, there are no Δ-fat paths in G^+ either. Therefore, a path cannot contribute more than Δ to the excess at the source.

By Lemma 6.7.8, the subgraph of G_g induced by the Δ-fat arcs has no flow-generating cycles. We know that $E^+ \subseteq E_g$, and therefore every flow-generating cycle in G^+ has at least one arc that is not Δ-fat. Consider a GAP in the decomposition and let (v, w) be a non Δ-fat arc along the flow-generating cycle in the GAP. The contribution of this GAP to the excess of the source is bounded by the fatness of the (v, s)-path in the GAP. The arc (v, w) is not Δ-fat, which implies that the fatness of this path is less than Δ. ▮

In Sections 6.7.2 and 6.7.3, we prove that the *Fat-Augmentation* and the *Cancel-Cycles* procedures can be implemented to run in $O(m + n \log n)$ and $O(mn^2 \log n \log B)$ time, respectively. Thus, we have the following theorem:

Theorem 6.7.3 The *Fat-Path* algorithm runs in $O(m^2n^2 \log n \log^2 B)$ time.

Proof: By Lemma 6.7.1, the algorithm terminates after $O(m \log B)$ phases. An augmentation either decreases the number of nodes with excess or increases the excess at the source by at least

```
Initialize: Start with an empty tree.
        For all $v \in V - s$, set $Gain(v) = 0$.
        Set $Gain(s) = 1$ and $Father(s) = s$.
Step 1  while there are nodes outside the tree with nonzero $Gain$ do
            a) Let $v$ be a node not in the tree with the maximum $Gain$.
            b) Add the arc $(v, Father(v))$ to the tree.
            c)For every neighbor $w$ of $v$ ($i.e.$, $(w,v) \in E$), if
              $u_g(w,v)\gamma_\mu(w,v)Gain(v) \geq \Delta$ and $Gain(w) < Gain(v)\gamma_\mu(w,v)$,
              then update $Gain(w) = Gain(v)\gamma_\mu(w,v)$ and $Father(w) = v$.
Step 2  Relabel the nodes in the tree using $\mu(v)Gain(v)$ as the new label of node $v$ (so that
        the relabeled gain of the tree arcs becomes 1).
Step 3  If there exists a node with excess in the tree, augment the current generalized
        pseudoflow from this node to $s$ along the unique path in the tree.
```

Figure 6.4: The Fat-Augmentation algorithm.

Δ. By Lemma 6.7.2, this gives an $O(m)$ bound on the number of augmentations per phase. ∎

6.7.2 Fat-Augmentation

The procedure *Fat-Augmentation*, shown in Figure 6.4, is a crucial part of the *Fat-Path* algorithm. It finds a node with excess such that the source is reachable from this node through a Δ-fat path, then finds the highest-gain Δ-fat augmenting path from this node to the source, and augments.

First we describe a simple (but not the most efficient) version of the algorithm. Consider a highest-gain augmenting path from a node v to the source. Either this path is Δ-fat, or the capacity of the arc (v, w) that would be saturated when the flow is augmented along this path, times the gain of the part of the path from v to the source, is below Δ. In this case we call (v, w) *critical.* The observation that a critical arc cannot be Δ-fat leads to a simple algorithm. First, find a highest-gain path from a node with excess, and check its fatness. If it is Δ-fat, we are done. If not, look for a critical arc, delete this arc from the set of arcs considered when searching for a highest-gain path, and repeat.

It is important to note that if an augmentation is done along the highest-gain Δ-fat path, then disregarded arcs do not become Δ-fat.

The above algorithm runs in polynomial time, but its running time can be improved. A faster method, proposed to us by Tarjan [personal communication], is the one described in Figure 6.4. The main idea is to recognize and disregard arcs that are not Δ-fat while constructing a highest-gain path. Let G_g^Δ denote the graph induced by the Δ-fat arcs in the residual graph of g. We search for a tree in G_g^Δ that is rooted at s, such that the path in this tree from any node to the source s has the highest gain among all Δ-fat paths from this node to the source. We call a tree with this property the "Max-Gain" tree. In order to improve the efficiency of finding this tree, we maintain a set of labels μ, such that all Δ-fat arcs have relabeled gains of at most 1. In particular, these labels guarantee that there are no flow-generating cycles in G_g^Δ.

The input to *Fat-Augmentation* is a parameter Δ, generalized pseudoflow g, and a set of labels μ, such that the relabeled gain γ_μ of arcs in G_g^Δ is at most 1. The output is a generalized pseudoflow g', together with updated labels μ', such that the relabeled gain of the arcs in $G_{g'}^\Delta$ with respect to μ' is at most 1.

The procedure can be viewed as a search for a shortest-path tree in G_g^Δ, where the length of an arc with relabeled gain γ_μ is $-\log \gamma_\mu$. There are no Δ-fat arcs with relabeled gain of above 1, and therefore the Δ-fat tree can be "grown" as in the Dijkstra's shortest-path algorithm.

Initially, the Δ-fat tree consists only of s. With each node v, we associate $Gain(v)$, which is the maximum gain of a Δ-fat path found so far from v to s. We initialize $Gain(v) = 0$, for all $v \in V - \{s\}$. Like in the Dijkstra's shortest path algorithm, at each iteration we find a node v that has the largest *Gain* among the nodes not in the tree, and add it to the tree. The difference is in the way we update the *Gain* of its neighbors. An arc (v, w) is disregarded by the algorithm if $u_g(v, w)Gain(w)\gamma_\mu(v, w) < \Delta$. We show below that an arc is disregarded if and only if it is not Δ-fat.

After the tree is constructed, the labels μ are updated, so that the relabeled gains of the arcs in the tree are equal to 1. Then we find a node in the tree with positive excess and augment the flow on the path from this node to the source.

To prove the correctness of the algorithm, we need to show that each augmentation is done only along a Δ-fat path, that the procedure finds a Δ-fat path if such a path exists, and that the subgraph of the Δ-fat arcs does not contain a flow-generating cycle.

Lemma 6.7.4

1. **The arcs used for updates in Step 1c are Δ-fat.**
2. **Augmentation is done on a Δ-fat path.**

Proof: Consider a path from some node to s in the tree constructed by the algorithm, and let (v, w) be a critical arc with respect to this path. This means that the fatness of this path is equal to $u_g(v, w)Gain(v) = u_g(v, w)Gain(w)\gamma_\mu(v, w)$. By construction, this is at least Δ, and hence the path is Δ-fat. A similar argument shows that all arcs used for updates are Δ-fat. ∎

Lemma 6.7.5 If all Δ-fat arcs have relabeled gains of less than or equal to 1 at the beginning of the procedure *Fat-Augmentation*, then Step 2 constructs a valid Max-Gain tree.

Proof: By Lemma 6.7.4, the arcs used for updating in Step 1c are all Δ-fat. Therefore the length $(-\log \gamma_\mu)$ of the arcs considered for updates is non-negative. Step 1 is an implementation of Dijkstra's shortest path algorithm on the graph induced in G_g by the considered arcs. Therefore the constructed tree is the shortest-path tree in this graph.

To prove the lemma, we have to show that disregarded arcs are not Δ-fat. Let (v, w) be the first Δ-fat arc disregarded by the algorithm. By definition there exists a path P_1 from w to s such that (v, w) and P_1 form a Δ-fat path. Denote by P_2 the path from w to s in the tree. The arc (v, w) is the first Δ-fat arc disregarded by the algorithm, and therefore, by construction, P_2 is the highest gain Δ-fat path. Moreover, we have

$$\begin{aligned} Gain(w) &= \prod_{e \in P_2} \gamma_\mu(e) \\ &\geq \prod_{e \in P_1} \gamma_\mu(e). \end{aligned}$$

The path (v, w) and P_1 is Δ-fat, and therefore $u_g(v, w)\gamma_\mu(v, w)\prod_{e \in P_1} \gamma_\mu(e) \geq \Delta$. Hence, $u_g(v, w)Gain(v)\gamma_\mu(v, w) \geq \Delta$. This contradiction shows that no Δ-fat arc will be disregarded by the algorithm. ∎

As an immediate corollary we get the following lemmas.

Lemma 6.7.6 **If the procedure did not augment, the source is not reachable in G_g^Δ from any node with excess.**

Lemma 6.7.7 **If all Δ-fat arcs have relabeled gain less than or equal to 1 at the beginning of an iteration of the procedure *Fat-Augmentation*, then the same is true after the relabeling in Step 2.**

Lemma 6.7.8 **In the graph $G_{g'}^\Delta$, where g' is the generalized pseudoflow g' returned by the procedure, there are no flow-generating cycles, and all arcs have relabeled gain of at most 1.**

Proof: The new labels μ are computed so that the augmentation is done along a path with relabeled gain 1. Therefore, the arcs whose residual capacity increases due to this augmentation all have relabeled gain 1. Since augmentation is done along the highest gain Δ-fat path, arcs that do not lie along this path do not become Δ-fat. ▮

The algorithm *Fat-Augmentation* can be implemented using Fibonacci heaps in a manner similar to the Fredman-Tarjan [43] implementation of the Dijkstra's shortest path algorithm.

Theorem 6.7.9 **The *Fat-Augmentation* algorithm runs in $O(m + n \log n)$ time.**

6.7.3 Canceling Flow-Generating Cycles

The aim of the algorithm described in this section is to convert a generalized pseudoflow g into a generalized pseudoflow g' whose residual graph has no flow-generating cycles, without decreasing the excess at the nodes. The idea is to cancel flow-generating cycles in the residual graph, while increasing the excesses at some nodes of the graph and without creating any deficits. The algorithm is an adaptation of Goldberg and Tarjan's [60] minimum cost flow algorithm that iteratively cancels negative-cost cycles in the residual graph.

Throughout this section, we refer to a generalized pseudoflow with no flow-generating cycles in the residual graph as *passive*. The Cycle-Canceling algorithm of Goldberg and Tarjan is based on the idea of ϵ-optimality. A circulation is said to be ϵ-optimal if the mean of the arc costs along any cycle in its residual graph, *i.e.*, the sum of the costs divided by the number of arcs in the cycle) is at least $-\epsilon$. To adapt this algorithm for generalized pseudoflows, we introduce the cost $c(v,w) = -\log\gamma(v,w)$ on the arcs. We say that a generalized pseudoflow is *ϵ-passive* if the mean cost of a cycle in its residual graph is at least $-\epsilon$. In other words, a generalized pseudoflow is ϵ-passive if the geometric mean of the gain-factors along every residual cycle is at most 2^{ϵ}. Given a generalized pseudoflow g, let ϵ_g denote the minimum ϵ such that g is ϵ-passive. The following lemma corresponds to Theorem 3.3 in [60].

Lemma 6.7.10 A generalized pseudoflow g can be relabeled so that the relabeled gain of any arc in the residual graph of g is at most 2^{ϵ_g}.

Proof: Define the length of an arc (v,w) in the residual graph to be $l(v,w) = -\log\gamma(v,w)+\epsilon_g$. By definition of ϵ_g, there are no negative-cost cycles with respect to this length. For $v \in V$, let $\rho(v)$ denote the length of the shortest path from v to s in the residual graph. (By Lemma 6.3.1 such path exist.) It can be seen that relabeling the graph using labels $\mu(v) = 2^{-\rho(v)}$ leads to a graph with the relabeled gain of every arc being below 2^{ϵ_g}. ▮

The following lemma corresponds to Theorem 3.2 in [60].

Lemma 6.7.11 If $\epsilon_g \leq 1/(nT)$, then any ϵ-passive generalized pseudoflow is passive.

Proof: Consider a cycle in the residual graph of a pseudoflow. By definition of T, if this cycle has gain above 1, then its gain is at least $1+T^{-1}$. This means that the mean cost of this cycle is at most $-(1/n)\log(1+T^{-1}) \leq 1/(nT)$. ▮

In the case of the minimum-cost flow problem, the Goldberg-Tarjan Cycle Canceling algorithm cancels cycles consisting of arcs with negative reduced cost. In the case of generalized pseudoflows we shall cancel cycles consisting of arcs with relabeled gain of above 1.

The algorithm is described in Figure 6.5. It starts with $\epsilon = \log B$ and proceeds in phases, where a phase consists of relabeling the graph so that the relabeled gain of every arc in the residual graph is at most 2^{ϵ} (see Lemma 6.7.10), and canceling all cycles in the graph induced by the arcs with relabeled gain above 1. Observe, that in the end of a phase any flow-generating cycle contains at least one arc with relabeled gain below 1, and the rest of the arcs have relabeled gain below 2^{ϵ}. Hence, we can set $\epsilon = (1-1/n)\epsilon$ and start the next phase.

Step 1 Relabel the residual network so that $\gamma(v,w) \leq 2^{\epsilon}$ for every (v,w) in the residual graph of the generalized pseudoflow g.

Step 2 Cancel all cycles in the residual graph of g where the relabeled gain of every arc is above 1.
Set $\epsilon = (1 - \frac{1}{n})\epsilon$.

Figure 6.5: A single phase of the *Cancel-Cycles* algorithm.

Lemma 6.7.12 The algorithm terminates in $O(\log T) = O(n \log B)$ phases, producing a generalized pseudoflow with no flow-generating cycles.

Proof: Immediate from Lemma 6.7.11 and the above discussion. ▮

Each time we cancel a cycle, we saturate at least one residual arc with relabeled gain above 1, and do not create any new such arcs. Therefore, we cancel at most m cycles during a single phase. Canceling a single flow-generating cycle, while creating excess at one of the nodes of the cycle, can be done in $O(n)$ time. By marking nodes that do not belong to cycles, we can easily obtain a running time of $O(nm)$ per phase, which leads to a total running time of $O(n^3 m \log B)$.

This running time can be improved using a variant of the Dynamic Tree data structure. Sleator and Tarjan introduced dynamic trees [119, 124, 120] in order to implement the operations described in Figure 6.6, where each operation takes amortized $O(\log n)$ time. This data structure was used for speeding up several maximum flow and minimum cost flow algorithms [119, 58, 63, 64, 60], including the Cycle-Canceling algorithm for minimum-cost flows. The main idea is that augmenting the flow along a path does not saturate all the arcs along this path. Instead, the path is subdivided into shorter paths. Storing these paths in a dynamic tree data structure allows to use the *add-value* operation to augment the flow in time which is logarithmic in the length of the path.

Unfortunately, this technique can not be applied directly for generalized circulations. The problem arises when we want to cancel a cycle. In order to do this, we must update the residual capacities of the arcs along the cycle. The amount of flow pushed along the arcs of the cycle is different in each arc and depends on the gains of the arcs. Therefore this update can not be done by subtracting the same value from all of them, as it is done in the case of flows.

make-tree(v): Make node v into a one-node dynamic tree. Node v must be in no other tree.

find-root(v): Find and return the root of the tree containing node v.

find-value(v): Find and return the value of the tree arc connecting v to its parent. If v is a tree root, return infinity.

find-min(v): Find and return the ancestor w of v such that the tree arc connecting w to its parent has minimum value along the path from v to *find-root*(v). In case of a tie, choose the node w closest to the tree root. If v is a tree root, return v.

change-value(v, x): Add real number x to the value of every arc along the path from v to *find-root*(v).

link(v, w, x): Combine the trees containing v and w by making w the parent of v and giving the new tree arc joining v and w the value x. This operation does nothing if v and w are in the same tree or if v is not a tree root.

cut(v): Break the tree containing v into two trees by deleting the arc from v to its parent. This operation does nothing if v is a tree root.

Figure 6.6: Dynamic tree operations.

We use a variant of the Dynamic Tree data structure, which we call the *Generalized Dynamic Tree* (GDT). The operations supported by this data structure are shown in Figure 6.7. The data structure is used to store and update the residual capacities of the arcs along the paths currently considered for augmentation. These arcs form a set of disjoint rooted trees. We shall show below that the Generalized Dynamic Tree operations can be implemented in $O(\log n)$ amortized time (Theorem 6.7.14).

Using Generalized Dynamic Trees, we can speed up Step 2 of the *Cancel-Cycles* algorithm to run in $O(m \log n)$ time. This implementation is shown in Figure 6.8. The algorithm picks a node v that is not yet marked as useless, and finds the root r of the tree that this node belongs to. If there are no arcs with relabeled gain above 1 from r to some node which was not yet marked as useless, we mark r as useless, remove it from the tree, and decompose the tree into a number of smaller trees. On the other hand, if there exists an arc (r, w) with relabeled gain above 1 and w was not yet marked, then w belongs to some tree. Let r' be the root of this tree. If r' and r are different nodes, then at this point we know that there is an augmenting path from v to r', that uses (r, w). In this case we insert (r, w) into our data structure by linking the two trees. If both v and w belong to the same tree, we have found a flow-generating cycle that consists of the arc (r, w) and the path from w to r in the tree. In this case we augment the flow along this cycle, removing from the tree all arcs whose residual capacity becomes zero.

make-tree(v): Make node v into a one-node dynamic tree. Node v must be in no other tree.

find-root(v): Find and return the root of the tree containing node v.

find-cap(v): Find and return the residual capacity of the tree arc connecting v to its parent. If v is a tree root, return infinity.

find-gain(v): Find and return the gain of the path in the tree connecting node v to to *find-root*(v).

find-sat(v): Find and return node w, such that the arc between w and its parent is the first one to get saturated if we increase the flow along the path from v to *find-root*(v). In case of a tie, choose the node w closest to the tree root. If v is a tree root, return v.

change-cap(v, x): Update the residual capacities of the arcs on the path from v to *find-root*(v). The residual capacity of each arc (w, w') on this path is decreased by x times the gain of the path from v to w in the tree.

link(v, w, x, γ): Combine the trees containing v and w by making w the parent of v and assigning x and γ to be the residual capacity and gain of this arc, respectively. This operation does nothing if v and w are in the same tree or if v is not a tree root.

cut(v): Break the tree containing v into two trees by deleting the arc from v to its parent. This operation does nothing if v is a tree root.

Figure 6.7: Generalized Dynamic Tree operations.

A straightforward analysis of the above procedure shows that it requires $O(m)$ tree operations. By Theorem 6.7.14, the use of Generalized Dynamic Trees leads to $O(m \log n)$ running time per phase. As it was already mentioned, after at most $O(n^2 \log B)$ phases the pseudoflow is passive, and therefore we have:

Theorem 6.7.13 The *Cycle-Canceling* algorithm runs in $O(mn^2 \log n \log B)$ time.

6.7.4 Implementing the Generalized Dynamic Tree Data Structure

The implementation of Generalized Dynamic Trees is based on two data structures. The first one is similar to the standard Dynamic Tree, and is used to store the gains on the arcs of the path. Using it, we can compute the gain of any path from a node to the root of the tree this node belongs to in amortized $O(\log n)$ time.

To implement the other operations that are supported by a Generalized Dynamic Tree data structure, we use a variant of Dynamic Trees that, in addition of supporting all the dynamic tree operations, supports the *multiply-value* operation that multiplies all the values stored in a tree

```
[Initialize]
for each node $v$, unmark $v$ and perform make-tree ($v$);
while there exists an unmarked node $v$ do begin
      let $r$ = find-root($v$);
      if there exists an arc $(r, w)$ in the residual graph with $\gamma_\mu(r, w) > 1$
      then begin
            if find-root ($w$) $\neq r$ then do [extend the path]
                              link($r, w, u_{g,\mu}(r, w), \gamma_\mu(r, w)$);
            else begin [cancel the cycle]
              $\delta = \min\{u_{g,\mu}(r, w)$, find-gain($w$)find-cap(find-sat($w$))$\}$;
              $g(r, w) = g(r, w) + \mu(r)\delta$;
              change-cap($w, \delta\gamma_\mu(r, w)$);
              while find-cap(find-sat($w$)) = 0 do begin
                    $z$ = find-sat($w$);
                    $g(z$, parent($z$)) = $u(z$, parent($z$));
                    cut($z$);
              end;
            end;
      else begin [remove $r$ from the path]
        mark $r$;
        for each node $z$ such that $r$ = parent($z$) do begin
          $g(z, r) = u(z, r) - \mu(z)$find-cap($z$);
          cut($z$);
        end;
      end;
end;
end.
```

Figure 6.8: Implementation of Step 2 of the *Cancel-Cycles* algorithm.

by a constant. This data structure is used to store residual capacities that are *relabeled to the root of the tree*. In other words, instead of the capacity of the arc (v, w), we store this capacity multiplied by the gain of the path from v to the root. The advantage of storing the capacities in this way is that changing the flow along a path changes the residual capacities of the arcs along the path by different amounts, dependent on the gains, whereas the relabeled capacities are changed by the same amount. The *multiply-value* operation is needed to relabel the capacities of the new trees to the new roots when we *cut* or *join* the trees. We will show (Theorem 6.7.15) that a sequence of dynamic trees operations, possibly including *multiply-value* operations, can be implemented in $O(\log N)$ amortized time, where N is the number of *make-tree* operations used. As a corollary, we get the following theorem.

Theorem 6.7.14 A sequence of M Generalized Dynamic Tree operations takes $O(M \log N)$ time,

where N is the number of *make-tree* operations in the sequence.

In order to be able to add the ***multiply-value*** operation to the standard dynamic tree operations, we change the way information is stored inside each element of the dynamic tree. Recall, that in a dynamic tree each tree is regarded as a collection of arc-disjoint paths. Dynamic tree operations are implemented in terms of operations on these paths. In order to implement a new operation, one has to add this operation to the set of operations supported by the underlying data structure that implements paths.

Each path in a Dynamic Tree is represented as a search tree, where the key is the distance from the end of the path. In the implementation suggested in [120] the search tree used is a splice tree. In order to distinguish between the trees in the graph (defined by the *link* and *cut* operations), and the search trees that represent paths, we call the latter trees *solid.*

Let $Minvalue(v)$ denote the minimum among the values of the descendants of v in a solid tree. Each node of a solid tree holds the following information:

$$\begin{aligned}\Delta value(v) &= value(v) - Minvalue(v)\\ \Delta min(v) &= \begin{cases} Minvalue(v) & \text{if } v \text{ is a solid tree root}\\ Minvalue(v) - Minvalue(parent(v)) & \text{otherwise.}\end{cases}\end{aligned}$$

Each operation on a path, say *find-minimum*(v), consists of scanning the solid tree representing the path from the node associated with v to the root of the tree. $Minvalue(v)$ can be computed by summing the values of Δmin in the nodes encountered during this scan. Using this representation, adding a value to all the capacities on a path requires adding this value to Δmin of the root of the solid tree, which takes constant time.

In order to be able to multiply the stored values by a constant without scanning all the nodes on the path, each node contains the following information instead of the above data:

$$\Psi value(v) = \begin{cases} \Delta value(v) & \text{if } v \text{ is a solid tree root,}\\ \Delta value(v)/\Delta value(parent(v)) & \text{otherwise.}\end{cases}$$

$$\Psi min(v) = \begin{cases} \Delta min(v) & \text{if } v \text{ is a solid tree root,}\\ \Delta min(v)/\Delta min(parent(v)) & \text{otherwise.}\end{cases}$$

It is easy to see, that by scanning the nodes on a path from a given node to the root of the solid tree, we can compute Δmin and $\Delta value$ of all the scanned nodes, and hence, we can also compute $Minvalue(v)$ and $value(v)$. More precisely, in order to be able to compute Δmin and $\Delta value$ even if some of them are zero, we keep the above ratios as tuples.

Observe that a change in the information stored in the root of a solid tree is sufficient in order to multiply $value(v)$ for every node in the solid tree by a constant. Adding a constant to $value(v)$ for all nodes v in the solid tree can be done by changing the information stored in the root of the solid tree and in its sons only. The rest of the operations on paths, required for the dynamic tree operations, are implemented exactly as for the standard dynamic trees.

Using the analysis of the Dynamic Tree data structure by Sleator and Tarjan either in [119] or in [120], it is straight-forward to see that storing $\Psi value$ and Ψmin instead of $\Delta value$ and Δmin in the nodes of the solid trees does not increase the running time. This means that it is possible to add the *multiply-value* operation to the set of the operations supported by dynamic trees without increase in the running time, which implies the following theorem.

Theorem 6.7.15 A sequence of M Dynamic Tree operations, possibly including *multiply-value* operations, takes $O(M \log N)$ time, where N is the number of *make-tree* operations in the sequence.

6.8 Conclusions

We have presented two polynomial-time combinatorial algorithms for the generalized circulation problem. The first algorithm is based on the repeated application of a minimum-cost flow subroutine; the second algorithm is based on the idea of augmenting along the biggest improvement path [36] and the idea of canceling negative cycles [60, 81]. Previous polynomial-time algorithms for the problem were based on general-purpose linear programming techniques, and the combinatorial structure of the problem was used solely for improving the efficiency of computing the required matrix inversions. Our results show that the problem can be handled by methods that are closer to the combinatorial methods traditionally used for network flow problems.

We introduce new tools for the design of combinatorial algorithms for the generalized circulation problem. Analysis of our algorithms is based on new insights into the comb'natorial structure of the problem and on deeper understanding of the relationship between the minimum cost flow problem and the generalized circulation problem. We believe that these tools and insights will lead to faster algorithms and to a better understanding of combinatorial structure of various network flow problems.

A by-product of our research is an increased appreciation of strongly polynomial algorithms. Our algorithms repeatedly use procedures to find shortest paths, maximum-flows, and minimum-cost flows. The input to these procedures may contain numbers which are much bigger than those in the input to the original problem. (See the remark at the end of the Section 6.6.) Because of this fact, we obtain better bounds by using strongly polynomial algorithms to solve these subproblems. This phenomena suggests that strongly polynomial algorithms may be important in practice as well as in theory: even though the numbers that occur in a statement of a problem may be relatively small, the numbers that occur in the intermediate problems can be large. This observation gives additional motivation to study strongly polynomial algorithms.

The methods used by Tardos [123] for designing a strongly polynomial algorithm for the minimum cost circulation problem were later extended by her in [122] for linear programs with integer constraint matrices. This yields an algorithm for the generalized circulation problem, whose running time is independent of the size of the capacities, and polynomial in n, m, and the number of bits needed to represent the gains. Our algorithms exploit a similarity between the gains and certain corresponding costs. Unfortunately, we were unable to use this similarity to extend the methods of [122] to construct a strongly polynomial algorithm for the generalized circulation problem.

Despite the apparent similarity between gains in the generalized circulation problem and costs in the minimum-cost circulation problem, the roles of these numbers are quite different. The difference stems from the fact that the gains appear in the constraint matrix of the corresponding linear program, whereas the costs appear only in the objective function.

In some applications, the generalized circulation problem is naturally stated by giving logarithms of gains in the input. For example, in the context of electrical networks, it is customary to use decibels to measure power loss in transmission lines. We call this representation of the problem the ***compact representation.*** This representation is also natural from the theoretical point of view, because of the intuitive correspondence between a generalized circulation problem with a gain function γ and the minimum-cost flow problem with the cost function $-\log\gamma$. On the compact representation, our algorithms do not run in polynomial time, and neither do the algorithms based on the linear programming techniques. In fact, all algorithms described in this chapter that run in polynomial time on the original representation of the problem run in pseudo-polynomial time (*i.e.*, in polynomial time if the input numbers are given in unary) on the compact representation. Solving the problem in polynomial time assuming the compact representation is closely related to finding a strongly polynomial algorithm for the generalized circulation problem. It is interesting to note that if the input numbers of the compact representation are given in unary, the simple Maximum Flow Based algorithm has the best time among the algorithms discussed in this chapter (see Corollary 6.5.3).

An important extension of the generalized circulation problem is the *generalized circulation with costs* problem. This problem has costs in addition to gains and capacities, and a fixed price $p(s)$ per unit of commodity of the source. The goal is to maximize profit, where the profit of a generalized circulation g is defined in a natural way. Vaidya's algorithm handles this extended problem with no modifications. Our algorithms, however, cannot handle this problem. It would be interesting to see if our algorithms can be modified to handle the generalized circulation with costs problem. A promising approach is to use the linear programming prices and reduced costs, as discussed above.

Bibliography

[1] K. Abrahamson. On acheiving consensus using shared memory. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, August 1988.

[2] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, 1987.

[3] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. Local management of a global resource in a communication network. In *Proc. 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 347–357, 1987.

[4] Y. Afek, G. M. Landau, B. Schieber, and M. Yung. The power of multimedia: combining point-to-point and multiaccess networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, August 1988.

[5] Y. Afek and M. Saks. Detecting global termination conditions in the face of uncertainty. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, August 1987.

[6] A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 325–334, 1987.

[7] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[8] R. K. Ahuja and J. B. Orlin. *New Distance-Directed Algorithms for Maximum-Flow and Parametric Maximum-Flow Problems*. Technical Report 1908-87, Sloan School of Management, M.I.T., 1987.

[9] R. J. Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 7, 1987.

[10] M. J. Atallah and S. R. Kosaraju. Graph problems on a mesh-connected processor array. *Journal of ACM*, 31(3):649–667, July 1984.

[11] B. Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32:804–823, 1985.

[12] B. Awerbuch. On the effects of feedback in dynamic network protocols. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, October 1988. (To appear.).

[13] B. Awerbuch. A tight lower bound on the time of distributed maximal independent set algorithms. February 1987. Unpublished manuscript.

[14] B. Awerbuch and S. Plotkin. *Approximating the size of a dynamically growing distributed network.* Technical Report MIT/LCS/TM-328, M.I.T., April 1987.

[15] P. Beame. 1987. Personal Communication.

[16] P. Beame. *Lower Bounds in Parallel Machine Computation.* PhD thesis, University of Toronto, 1986.

[17] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[18] D. P. Bertsekas. *Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems.* Technical Report LIDS-P-1986, Lab. for Decision Systems, M.I.T., September 1986. (Revised November, 1986).

[19] G. Blelloch. *Parallel Prefix vs. Concurrent Memory Access.* Technical Report, Thinking Machines, Inc., 1986.

[20] B. Bloom. Constructing two-writer atomic registers. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[21] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *J. Computer and System Sci.*, 30:130–145, 1985.

[22] J. Boyar and H. Karloff. Coloring planar graphs in parallel. *J. of Algorithms*, (8):470–479, 1987.

[23] R. G. Busacker and P. J. Gowen. *A Procedure for Determinimg a Family of Minimal-Cost Network Flow Patterns.* Technical Report 15, O.R.O., 1961.

[24] R. G. Busacker and T. L. Saaty. *Finite Graphs and Networks: An Introduction with Applications.* McGraw-Hill, New York, NY., 1965.

[25] A. K. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, May 1984.

[26] N. Chiba, T. Nishizeki, and N. Saito. A linear 5-color algorithm of planar graphs. *Journal of Algorithms*, 2:317–327, 1981.

[27] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, January 1987.

[28] T. W. Christopher. *An implementation of Warshall's algorithm for transitive closure on a cellular computer.* Technical Report 36, Inst. for Comp. Research, Univ. of Chicago, 1973.

[29] M. Chrobak, K. Diks, and T. Hagerup. Parallel 5-coloring of planar graphs. In 14^{th} *International Colloquium on Automata, Languages, and Programming*, pages 304–313, July 1987.

[30] B. A. Coan and R. Turpin. *Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement.* Technical Report MIT/LCS/TR-315, M.I.T., Laboratory for Computer Science, 1984.

[31] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–56, 1986.

[32] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, November 1981.

[33] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11:1–4, August 1980.

[34] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[35] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[36] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[37] J. Elam, F. Glover, and D. Klingman. A strongly convergent primal simplex algorithm for generalized networks. *Math. of O. R.*, 4:39–59, 1979.

[38] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[39] M. J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[40] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[41] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ., 1962.

[42] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[43] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[44] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial time hierarchy. In *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, pages 260–270, 1981.

[45] H. N. Gabow and R. E. Tarjan. Almost-optimal speed-ups of algorithms for matching and related problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 514–527, 1988.

[46] R. G. Gallager, P. A. Humblet, and P. M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

[47] R. K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:134–150, 1984.

[48] F. Glover, J. Hultz, D. Klingman, and J. Stutz. Generalized networks: a fundamental computer-based planning tool. *Management Science*, 24(12), August 1978.

[49] F. Glover and D. Klingman. On the equivalence of some generalized network problems to pure network problems. *Math. Programming*, 4:269–278, 1973.

[50] A. Goldberg and S. Plotkin. *Efficient Parallel Algorithms for* $(\Delta+1)$*-Coloring and Maximal Independent Set Problems.* Technical Report MIT/LCS/TM-320, M.I.T., January 1987.

[51] A. Goldberg and S. Plotkin. Parallel $(\Delta + 1)$ coloring of constant-degree graphs. *Information Processing Letters*, 25(4):241–245, June 1987.

[52] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. *SIAM J. on Descrete Mathematics*, 1988. To appear.

[53] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. In *Proc. 19th ACM Symposium on the Theory of Computing*, pages 315–324, May 1987.

[54] A. Goldberg, S. Plotkin, and É. Tardos. Combinatorial algorithms for the generalized circulation problem. In *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, October 1988. (to appear).

[55] A. Goldberg, S. Plotkin, and É. Tardos. *Combinatorial Algorithms for the Generalized Circulation Problem.* Technical Report MIT/LCS/TM-358, M.I.T., June 1988.

[56] A. Goldberg, S. Plotkin, and P. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, October 1988. (to appear).

[57] A. Goldberg, S. Plotkin, and P. Vaidya. *Sublinear-Time Parallel Algorithms for Matching and Related Problems.* Technical Report MIT/LCS/TM-357, M.I.T., June 1988.

[58] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[59] A. V. Goldberg. *A New Max-Flow Algorithm.* Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[60] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 388–397, 1988.

[61] A. V. Goldberg and R. E. Tarjan. *Finding Minimum-Cost Circulations by Successive Approximation.* Technical Report MIT/LCS/TM-333, Laboratory for Computer Science, M.I.T., 1987. Also available as Technical Report CS-TR 106-87, Department of Computer Science, Princeton University.

[62] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 136–146, 1986. (To appear in J. Assoc. Comput. Mach.).

[63] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *JACM*, (To appear).

[64] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 7–18, 1987.

[65] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. In *Proc. 28th IEEE Symp. on Foundations of Comp. Sci.*, pages 161–165, 1987.

[66] M. Gondran and M. Minoux. *Graphs and Algorithms.* Wiley, 1984.

[67] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation for combinatorial algorithms. In *Proc. of the Caltech Conference on Very Large Scale Integration*, pages 509–525, January 1979.

[68] F. Harary. *Graph Theory.* Addison-Wesley, 1972.

[69] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, August 1988.

[70] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, January 1987.

[71] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[72] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Information Proc. Lett.*, 22:57–60, 1986.

[73] J.J. Jarvis and A.M. Jezior. Maximal flow with gains through a special network. *Operations Res.*, 20:678–688, 1972.

[74] W. S. Jewell. *Optimal Flow through Networks.* Technical Report 8, M.I.T., 1958.

[75] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 147–159, 1986.

[76] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[77] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a maximum matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[78] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the Association for Computing Machinery*, 32(4):762–773, October 1985.

[79] L. G. Khachian. Polynomial algorithms in linear programming. *Zhurnal Vychislitelnoi Matematiki i Matematicheskoi Fiziki*, 20:53–72, 1980.

[80] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41, Princeton University Press, 1956.

[81] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14:205–220, 1967.

[82] P. Klein and J. Reif. An efficient parallel algorithm for planarity. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, pages 465–477, 1986.

[83] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. Assoc. Comp. Mach.*, 27:831–838, 1980.

[84] L. Lamport. *On Interprocess Communication, Parts I and II.* Technical Report 8, Digital, System Research Center, December 1985.

[85] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[86] F. T. Leighton. Introduction to the theory of networks, parallel computation and VLSI design. 1988. Unpublished manuscript.

[87] C. Leiserson and B. Maggs. Communication-efficient parallel graph algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

[88] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. Optimization of synchronous circuitry by retiming. In *Third Caltech Conference on VLSI*, pages 87–116, March 1983.

[89] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–46, Spring 1983.

[90] K. N. Levitt and W. H. Kautz. Cellular arrays for the solution of graph problems. *Communications of the ACM*, 15(9):789–801, September 1972.

[91] N. Linial. Locality as an obstacle to distributed computing. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 331–335, October 1987.

[92] M. C. Loui and H.H. Abu-Amara. *Advances In Computing Research.* Jai Press, 1987.

[93] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Comp.*, 15(4):1036–1052, November 1986.

[94] N. A. Lynch, N. D. Griffeth, M. J. Fisher, and L. J. Guibas. Probabilistic analysis of a network resource allocation algorithm. *Information and Control*, 68:47–85, 1986.

[95] N. A. Lynch and E.W. Stark. *A Proof of the Kahn Principle for Input/Output Automata.* Technical Report MIT/LCS/TM-349, M.I.T., Laboratory for Computer Science, January 1988.

[96] N. A. Lynch and M.R. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* Technical Report MIT/LCS/TR-387, M.I.T., Laboratory for Computer Science, April 1987.

[97] B. Maggs and S. Plotkin. Minimum-cost spanning tree as a path-finding problem in a closed semiring. *Information Processing Letters*, 26(6):291–293, January 1988.

[98] D. Matula, Y. Shiloach, and R. Tarjan. *Two Linear-time Algorithms for Five-coloring a Planar Graph.* Technical Report STAN-CS-80-830, Department of Computer Science, Stanford University, Palo Alto, California, November 1980.

[99] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Computers*, 9(1):39–47, 1960.

[100] N. Megiddo. Towards a genuinely polynomial algorithm for linear programming. *SIAM J. Comput.*, 12:347–353, 1983.

[101] G. Miller and J. Reif. Parallel tree contraction and its application. In *Proc. of 26th IEEE Symp. on Foundations of Computer Science*, pages 478–489, October 1985.

[102] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 345–354, 1987.

[103] J. Naor. *Two Parallel Algorithms in Graph Theory.* Technical Report CS-86-6, Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, June 1986.

[104] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 232–249, 1987.

[105] K. Onaga. Dynamic programming of optimum flows in lossy communication nets. *IEEE Trans. Circuit Theory*, 13:308–327, 1966.

[106] K. Onaga. Optimal flows in general communication networks. *J. Franklin Inst.*, 283:308–327, 1967.

[107] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proc. 20th ACM Symp. on Theory of Computing*, 1988. 377-387.

[108] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proc. 17th ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[109] G.L. Peterson and J.E. Burns. Concurrent reading while writing. In *Proc. 28th IEEE annual Symp. on Foundation of Computer Science*, pages 383–392, 1987.

[110] G.L. Peterson and J.E. Burns. Constructing multi-reader atomic values from non-atomic values. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987.

[111] J. C. Picard and M. Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12:141–159, 1982.

[112] P. S. Pulat and S. E. Elmaghraby. *On Maximizing Flow in Generalized Flow Networks.* Technical Report 202, NC State University, 1984.

[113] J. Reif. An optimal parallel algorithm for integer sorting. In *Proc. of 26th IEEE Symp. on Foundations of Computer Science*, pages 496–503, October 1985.

[114] R. Schaffer. *On the Correctness of Atomic Multi-Writer Registers.* Technical Report MIT/LCS/TM-364, M.I.T., Laboratory for Computer Science, July 1988. Edited by B. Bloom.

[115] B. Schieber and S. Moran. Slowing down sequential algorithms for obtaining fast distributed and parallel algorithms: maximum matchings. In *Proc. 5th ACM Symp. on Principles of Distributed Computing*, August 1986.

[116] F. L. Van Scoy. The parallel recognition of classes of graphs. *IEEE Transactions on Computers*, C-29(7):563–570, July 1980.

[117] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[118] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[119] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[120] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32:652–686, 1985.

[121] J. Smith. Parallel algorithms for depth-first searches I. Planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

[122] É. Tardos. A strongly polynomial algorithm for solving combinatorial linear programs. *Operations Research*, 250–256, 1986.

[123] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[124] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[125] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput*, 1:146–160, 1972.

[126] K. Truemper. On max flows with gains and pure min-cost flows. *SIAM J. Appl. Math.*, 32:450–456, 1977.

[127] P. M. Vaidya. An algorithm for linear programming that requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ arithmetic operations. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 29–38, 1987.

[128] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware In *Proc. 27th IEEE annual Symp. on Foundation of Computer Science*, pages 233–243, 1986.

[129] S. Warshall. A theorem on boolean matrices. *Journal of ACM*, 9(1):11–12, 1962.

OFFICIAL DISTRIBUTION LIST

Director 2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 6 copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center 12 copies
Cameron Station
Alexandria, VA 22314

National Science Foundation 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 1 copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555